ON THE ALGEBRAIC DENOTATIONAL SPECIFICATIONS OF PROGRAMMING LANGUAGE SEMANTICS

by

JUEMIN SUN

B.S., Fudan University (China), 1984

A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Dept. of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas
1988

Approved by:

Major Professor

LD
2668
.T4
CMSC
1988
S86
c. 2

A11208 135672

Contents

Acknowledgements

I am deeply grateful to my advisor Prof. Maria Zamfir-Bleyberg who made this project possible to me and whose invaluable suggestions helped this paper shaped. Very very special thanks to Prof. Dave Schmidt not only for his correction of several mistakes in the draft of this paper but also for his excellent courses of CMPSC 705 and CMPSC 806 that introduced me to the field of programming language semantics. I also wish to thank the other committee member Prof. Bill Hankley with whom I had interesting course of CMPSC 671 that aroused my interests in the area of formal specifications.

Needless to say, I am most grateful to our computer science department for providing such good research environment and giving me financial support in the form of Graduate Teaching Assistantship.

I would like to dedicate this work to my wife Yuhe. Without her support and understanding this paper would simply not have resulted.

Chapter 1
Introduction

Since the years around 1970 mainly two general directions of research in the areas of software development can be distinguished. These two directions are mathematical semantics of programming languages and rigorous approaches to abstract data types in programming and specification languages. Denotational semantics as developed by Scott and Strachey is the most prominent approach of the first direction. Many-sorted algebras and their specification in terms of equations are the mathematical fundament of the second direction of research. Denotational semantics and many-sorted algebras form the background of this paper.

The above two general directions of research in computer science, i.e., mathematical semantics and abstract data type specification, have influenced each other and can not be separated. In the Scott-Strachey approach to programming language semantics ([St 77] [Sc 86]), a language is given a semantics by mapping each its syntactic constructs into a set of mathematical domains (called Scott domains). The hope is that one will be able to reason about programming language constructs by using the properties of these domains. Denotational semantics has been used to describe various programming languages by giving models based on higher order functions on Scott domains. Techniques have been developed for representing features of programming languages in terms of the basic operations of λ-notation: application, abstraction, tupling and tagging. The denotational semantics approach has led to the so called initial algebra semantics of programming languages which originates in the work of algebraic specification of abstract data types by [GTWW 77]. A fundamental tenet of the initial algebra semantics is that syntactic constructs reside in initial objects and that semantics is completely determined by specifying an algebra with the same signature as the

syntactic algebra and by specifying the values of the generators; then the semantic function is the unique homomorphism from the syntactic algebra to the semantic one.

In this paper we describe systematically a method for giving algebraic denotational specifications of programming language semantics in OBJ3, a first-order parameterized algebraic specification language. OBJ3 has an underlying mathematical semantics that is based on the initial (order-sorted) algebra semantics, and an operational semantics that is based on order-sorted rewriting rules. Our definitions are denotational in the sense that meanings are elements of particular abstract data types, i.e., the initial algebras of the specifications of denotational semantic domains. The structure of specifications has direct connection with that of denotational semantics of [Sc 86]. Our specifications of programming language semantics consist of three kinds of OBJ3 modules: the object SYN of syntactic domains that is based on abstract syntax, the object(s) SEM of semantic domains that are used as meanings in the definitions, and the object VAL of valuation functions that contains both objects of SYN and SEM and additional operations that map elements of SYN to those of SEM. All modules of these three kinds construct a directed acyclic graph.

The main difference between our specifications and standard denotational semantics is that we use first-order algebraic specification language OBJ3 instead of high order λ-functions as metalanguages in definitions. Since OBJ3 has an underlying fixed semantics that is based on the initial algebra semantics, we have to use the initial algebras of specifications as semantic domains, while in denotational semantics Scott domains (or cpos) are used. The major effort of this paper is to study the specifications of higher order domains in the first-order algebraic language.

Our definitions have the advantage of being immediately executable. It is often found that even fairly simple definitions are usually wrong as first written and need to be debugged in the same way that programs

are. This is especially true for the students who are beginning to learn denotational semantics. It would be of great help to run test cases when the definitions are executable. As another advantage, although our definitions are denotational in nature, we also have the benefits of using algebraic techniques. Our definitions are highly structured with increased flexibility and easy verifiability. The modularization and parameterization mechanisms of OBJ3 ensure that our specifications of semantic domains are maximally reusable in definitions of other programming languages.

[GP 81] described a similar method for giving structured algebraic denotational definitions of programming language semantics. They used algebraic specification language OBJT, which is based on the error algebras, and gave a semantic specification for a modest block-structured language. The basic idea is to use parameterized abstract data types to construct a directed asyclic graph of modules, such that each module corresponds to some *feature* of the language. A "feature" in their sense is sometimes a syntactic construction, and is sometimes a more basic language design decision. The major difference with our method is that our specifications are closer to the structure of standard denotational semantics, and we use a better version of algebraic specification language: OBJ3, which is based on the order-sorted algebras. In this paper, the issues of using first-order algebraic language to specify denotational semantics are more thoroughly dealt with. We cover such topics as high order function domain specifications, recursively-defined domain specifications, and continuation domain specifications that are not mentioned in [GP 81].

Now we outline the structure of this paper. After the introduction of this chapter, chapter 2 will briefly cover the basic concept of many sorted algebras and its extension to the order-sorted algebras. The materials in this chapter are the semantic foundation of OBJ3, and are quite independent from the rest of the paper. One with some knowledge of initial algebra semantics may safely skip this chapter. Chapter 3

introduces language features of OBJ3. Since OBJ3 is still under development, we only cover those features that have successfully worked in our specifications. Many seemingly promising specifications but failed to work in the current version of OBJ3 are not included in this paper. Chapters 4 and 5 are the main part of this paper. In chapter 4, after a brief introduction to the components of denotational semantics, the structure of our algebraic denotational specification is described. This chapter emphasizes the specification of syntactic domains from given abstract syntax. Problems of using algebras as semantic domains and decurrying transformations are also discussed. In chapter 5 we concentrate on the specification of semantic domains, especially higher order domains. In particular, we describe two methods of giving first order specifications for the function domains: defunctionalization and lambda-calculus. Throughout this chapter, we explain in detail specifications of three programming languages: BLOK1, PLISP, and BLOK2, which are included in the appendixes. BLOK1 is a strong-typing, block-structured language that has conditional and repetition commands. A specification of direct semantics is given for BLOK1. PLISP is a pure LISP-like language adapted from [Sc 86]. The specification of this language illustrates the method of specifying recursively defined domains. The third language BLOK2 is similar to BLOK1. We present a specification of continuation-based semantics for it.

Chapter 2
OBJ3 Semantic Foundations

OBJ3 is a wide spectrum, first order functional programming language with an underlying formal semantics that is based on initial algebra semantics in particular order-sorted algebras, and an operational semantics that is based on order-sorted rewrite rules. This rigorous semantic basis allows a declarative, specificational style of programming, eases system design and implementation, and facilitates program verification.

This chapter covers the initial algebra semantics and equational deduction. Since order-sorted algebra (OSA) is a generalization of many-sorted algebra, sections 2.1. and 2.2. introduce the basic concepts of MSA and equational deduction. Section 2.3. then gives a brief description of this generalization to OSA. For technical details of OSA and order-sorted deduction, one is referred to [GM 88] and up-coming consecutive papers on the subjects of OBJ3 semantic foundation.

Initiality was developed in category theory, where it is one of the most elementary concepts, and first entered computer science in an algebraic approach to abstract syntax and compositional semantics [GTWW 77]. The first great success of initial algebra semantics was Abstract Data Types (ADT), for which it gave the first rigorous semantics [GTW 78].

2.1. Many-Sorted Algebra

A many-sorted algebra (MSA) has several sets, called the *carriers* of the algebra, together with an indexed family of operations from Cartesian products of those carriers into one of them. The carriers are indexed by a set S, called the set of sorts. The following accounts of MSA are based on [GTW 78], [GM 86], [BL 79], [EM 85] and [NR 85]. We simply present definitions and important theorems, all proofs are

omitted.

2.1.1. Signatures and Algebras

Definition 1: An S-sorted signature (S,Σ) consists of a set S, called the set of sorts and an $S^*\times S$-indexed family $\langle \Sigma_{w,s} \mid w\in S^*,\ s\in S \rangle$ of disjoint sets. $\sigma\in\Sigma_{w,s}$ is an operator symbol of arity w and sort s; the pair $\langle w,s\rangle$ is called the rank of σ.

The arity of an operator symbol specifies what sorts of data it expects to see as inputs and in what order; and the sort of an operator symbol specifies the sort of data it returns. A constant symbol of sort s has arity the empty string λ; i.e. it is a member of $\Sigma_{\lambda,s}$.

Example: A signature Σ for the natural numbers might have S={nat,bool} with $\Sigma_{\lambda,bool}=\{T,F\}$, $\Sigma_{\lambda,nat}=\{0\}$, $\Sigma_{nat,nat}=\{inc\}$, $\Sigma_{nat,bool}=\{odd\}$, $\Sigma_{nat\ nat,\ nat}=\{+\}$, and $\Sigma_{w,s}=\emptyset$ otherwise.

A signature Σ says nothing about how to interpret the sorts as actual sets of data elements and the operator symbols as actual operators. Each such interpretation is called a Σ-algebra. Many different algebras may have the same signature.

Definition 2: A Σ-algebra A consists of an S-indexed set $\langle A_s \mid s \in S\rangle$ of carrier sets, and for each operator symbols σ in $\Sigma_{w,s}$ an actual operator $\alpha(\sigma):A^w\rightarrow A$ where $A^w=A_{s1}\times\ldots\times A_{sn}$ when w=s1...sn (when $w=\lambda$, then A^w is a one point set).

Notice that α is an $S^*\times S$-indexed set of interpretation mappings

$$\alpha_{w,s}: \Sigma_{w,s}\rightarrow[A_w\rightarrow A]$$

for the operator symbols in Σ, each $\alpha_{w,s}$ interpreting σ in $\Sigma_{w,s}$ as a function from A^w to A_s. It is usual to write σ for $\alpha(\sigma)$ if the algebra in question is clear from context, and it is normally more convenient to write σ_A if it isn't.

2.1.2. Homomorphisms and Isomorphisms

definition 3: If A and B are both Σ-algebras, a **Σ-homomorphism** h: A→B is an S-indexed function, i.e. a family of functions $\langle h_s : A_s \to B_s \mid s \in S \rangle$, that preserves the operations:

(h0) If $\sigma \in \Sigma_{\lambda,s}$, then $h_s(\sigma_A) = \sigma_B$;

(h1) if $\sigma \in \Sigma_{s1 \ldots sn,s}$ and $\langle a1,\ldots,an \rangle \in A_{s1} \times \ldots \times A_{sn}$, then $h_s[\sigma_A(a1,\ldots,an)] = \sigma_B[h_{s1}(a1),\ldots,h_{sn}(an)]$.

The composite of homomorphisms is again a homomorphism; composition is an associative operation. The identity function, I_A, on the carriers of A (so actually the S-indexed family of identity functions) is a Σ-homomorphism which is the identity for composition.

Definition 4: A Σ-homomorphism f: A→B is a **Σ-isomorphism** if and only if there is another Σ-homomorphism g: B→A such that $f \bullet g = I_A$ and $g \bullet f = I_B$; this g is unique if it exists, called **inverse** of f.

2.1.3. Initial Algebras

Definition 5: A Σ-algebra A is **initial** in a class C of Σ-algebras if and only if for every Σ-algebra B in C there exists a *unique* Σ-homomorphism h: A→B.

One common case is that C is the class of all Σ-algebras; another is that C is the class of all Σ-algebras that satisfy some set E of equations (then C is called the **variety** of E). In general, the class C is not mentioned when it is clear from context.

The tremendous usefulness of this definition is embodied in the following theorem.

Theorem 6: Let A be initial in a class C of Σ-algebras. Then an algebra A′ is also initial in C if and only if A is Σ-isomorphic to A′; in fact, there is a unique Σ-isomorphism from A to A′.

It is standard practice in algebraic semantics to "identify" isomorphic objects, that is, to treat them as identical. Thus we may speak of "*the* initial algebra" in a class of Σ-algebras C, for any two initial algebras are isomorphic, and there is a unique isomorphism from one to

the other. The wonderful thing about initiality is that it characterizes uniquely up to isomorphism; that is, it provides an abstract characterization, up to isomorphism. This observation is very important in the study of abstract data type [GTW 78].

2.1.4. Term Algebras, Initial and Free Algebras

One thing we obviously need is a general existence theorem for initial algebras; we want to know that these objects exist and something about what they look like. In this subsection, we will give an *term algebra* (or *Herbrand universe*) construction for an initial Σ-algebra T_Σ by mutual recursion among sets of Σ-terms. To be general enough, given a S-sorted signature Σ we directly start to define the S-indexed family of Σ-terms with *variables* (or *generators*) from an indexed family of sets, $X = \langle X_s \mid s \in S \rangle$. This will give us the carrier of the Σ-algebra freely generated by X.

We assume that the sets X_s are pairwise disjoint and also disjoint with $\Sigma_{w,s}$. The union $X = \bigcup_{s \in S} X_s$ is called **set of variables w.r.t. Σ.**

Given a S-sorted signature Σ and the set of variables X, we have

Definition 7: The S-indexed set of **Σ-terms** $T_\Sigma(X) = \langle T_{\Sigma,s}(X) \mid s \in S \rangle$ is recursively defined (over the set $\Sigma \cup X \cup \{(,)\}$, here Σ ambiguously denotes the set of all operator symbols in the S-indexed signature Σ, i.e. $\cup\{\Sigma_{w,s} \mid w \in S^*, s \in S\}$) by:

(1) $X_s \cup \Sigma_{\lambda,s} \subseteq T_{\Sigma,s}(X)$;

(2) if $\sigma \in \Sigma_{s1 \ldots sn,s}$ and $ti \in T_{\Sigma,si}(X)$ then $\sigma(t1 \ldots tn) \in T_{\Sigma,s}(X)$.

Now we make $T_\Sigma(X)$ into a Σ-algebra by defining the operations σ_T for each operator symbols in the signature Σ.

Definition 8: (1) For $\sigma \in \Sigma_{\lambda,s}$, $\sigma_T = \sigma \in T_{\Sigma,s}(X)$.

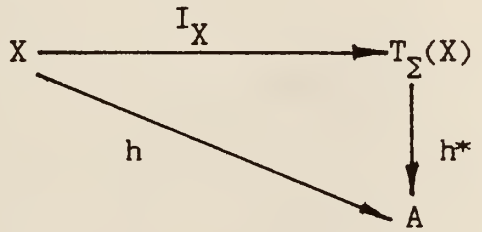
(2) For $\sigma \in \Sigma_{s1 \ldots sn,s}$ and $ti \in T_{\Sigma,si}(X)$,

$\sigma_T(t1, \ldots, tn) = \sigma(t1 \ldots tn) \in T_{\Sigma,s}(X)$.

Then we have the following theorem:

Theorem 9: Let I_X: $X \rightarrow T_\Sigma(X)$ be the S-indexed family of set injections of of the variables X into the carrier of $T_\Sigma(X)$. Then $\langle I_X, T_\Sigma(X)\rangle$ is the algebra **freely generated** by X in the class of Σ-algebras. That is, for any Σ-algebra A and any map h: $X \rightarrow A$ (again, S-indexed family) there exists a unique Σ-homomorphism h^*: $T_\Sigma(X) \rightarrow A$ such that $I_X h = h^*$.

i.e. the following diagram commutes:



The essential result is as follows:

Theorem 10: $T_\Sigma(\emptyset) = T_\Sigma$ is the initial Σ-algebra.

2.1.5. Equations

The initial algebra T_Σ is sometimes called the *anarchic* Σ-algebra, since it obeys no laws at all. It provides only a beginning point because we want to consider initial (and free) algebras in the class of algebras which are constrained to satisfy certain "laws" or "axioms" or "equations". The carriers of such initial Σ-algebras that satisfy certain equations will consist of equivalence classes of Σ-terms. We use $T_\Sigma(X)$ as syntax for presenting classes of algebras satisfying certain properties.

Definition 11: A **Σ-equation** of sort s is a pair $e=\langle t1, t2\rangle$, where t1, t2 are in $T_{\Sigma,s}(X)$. An **equational system** over $T_\Sigma(X)$ is a set (family) E of Σ-equations.

Definition 12: Given a Σ-algebra A, an **assignment** is a mapping from X to to A: $f: X \rightarrow A$.

By the Theorem 9, there is a unique Σ-homomorphism from $T_\Sigma(X)$ to A, i.e., a unique Σ-homomorphism $T_\Sigma(X) \rightarrow A$ extending f; let us denote it f*. we now have the following two definitions:

Definition 13: A Σ-algebra A **satisfies** the Σ-equation e=<t1, t2> if and only if for every assignment f: $X \rightarrow A$, f*(t1)=f*(t2). Given an equational system E, A satisfies E if and only if A satisfies each equation in E; in that case, A is called a **(Σ,E)-algebra**.

Definition 14: The pair (Σ,E) is called an **equational presentation**; and the **variety** of E is the class of all (Σ,E)-algebras.

The following generalization of Theorem 6 says that there always are initial (Σ,E)-algebras.

Theorem 15: For any signature Σ and equational system E, there is an initial (Σ,E)-algebra.

2.1.6. Congruences and Quotients

Now we proceed to obtain the initial (Σ,E)-algebra, hereafter denoted $T_{\Sigma,E}$. To do this we must have the following definition.

Definition 16: A **Σ-congruence**, ≡ on a Σ-algebra A is a family $\langle \equiv_s \mid s \in S \rangle$ of equivalence relations, $\equiv_s$ on A_s, with the *substitution property*: for all $\sigma \in \Sigma_{s1\ldots sn,s}$, if $ai,bi \in A$ and if $ai \equiv_s bi$ $(1 \leq i \leq n)$ then $\sigma_A(a1,\ldots,an) \equiv_s \sigma_A(b1,\ldots,bn)$

If A is Σ-algebra and ≡ is a Σ-congruence on A, let A/≡ be the S-indexed family of sets equivalence classes, $A/\equiv = \langle A_s/\equiv_s \mid s \in S \rangle$. Let $[a]_s$ (or just [a]) be the equivalence class of $s \in A_s$. We now make A/≡ into a Σ-algebra by defining the operations $\sigma_{A/\equiv}$.

(1) if $\sigma \in \Sigma_{\lambda,s}$, then $\sigma_{A/\equiv} = [\sigma_A]$.

(2) if $\sigma \in \Sigma_{s1\ldots sn,s}$ and $[ai] \in (A/\equiv)_{si}$ then

$$\sigma_{A/\equiv}([a1],\ldots,[an]) = [\sigma_A(a1,\ldots,an)].$$

Theorem 18: The A/≡ defined above is a Σ-algebra, called the **quotient** of

A by $\equiv$.

The initial algebra $T_{\Sigma,E}$ is a quotient of T_Σ by a congruence relation obtained from E. To make this precise, we first define the "congruence relation generated by an (arbitrary) relation" on an algebra.

Theorem 19: Let A be Σ-algebra, and let R be a relation on A. Then there is a least Σ-congruence relation on A containing R; it is called the *congruence relation generated by R* on A.

The proof (see chapter 3 of [EM 85]) is highly nonconstructive and gives no hint about how to determine whether some pair $\langle a,a' \rangle$ is in the congruence. In the following we give a construction of a congruence from a Σ-algebra A and an equational system E over Σ.

Theorem 20: Given a Σ-presentation (Σ,E) and a Σ-algebra A, the following inductively defined family $\equiv_E$ is a congruence on A:

(1) $h^*(L) \equiv_E h^*(R)$ for all $(L,R) \in E$ and $h: X \rightarrow A$. where h^* denote the unique homomorphic extension of h.

(2) $a \equiv_E a$, for all $a \in A_s$ and $s \in S$.

(3) if $a \equiv_E b$ then $b \equiv_E a$, for all $a,b \in A_s$ and $s \in S$.

(4) if $a \equiv_E b$, $b \equiv_E c$ then $a \equiv_E c$, for all $a,b,c \in A_s$ and $s \in S$.

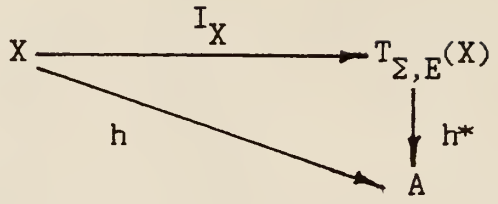
(5) $\equiv_E$ has the *substitution property* (as in definition 16).

Moreover, $\equiv_E$ is the smallest congruence on A which satisfies property (1).

We define $T_{\Sigma,E}(X)$ to be $T_\Sigma(X)/\equiv_E$, the quotient of the free Σ-algebra by the congruence relation $\equiv_E$ constructed above. Let I_X: $X \rightarrow T_{\Sigma,E}(X)$ be the canonical map, $I_X: x \rightarrow [x]$, taking each x to the congruence class of x relative to $\equiv_E$.

Theorem 21: $\langle I_X, T_{\Sigma,E}(X) \rangle$ is the algebra freely generated by X in the class of all (Σ,E)-algebras: for any (Σ,E)-algebra A and for any set map $h: X \rightarrow A$, there exists a unique homomorphism h^*: $T_{\Sigma,E}(X) \rightarrow A$ such

that the following diagram commutes:



Similarly to Theorem 10, we have

Theorem 22: $T_{\Sigma,E} = T_{\Sigma,E}(\emptyset)$ is the initial (Σ,E)-algebra.

2.2. Equational Deduction and Term Rewriting

In this section, we describe a set of rules for equational deduction that is sound and complete. Soundness means that applying the rules to a given set of equations always yields equations that are satisfied by any algebra that satisfies the original equations. And completeness means that every equation satisfied by all algebras satisfying the given equations can be deduced using the rules. In 2.2.3 term rewriting with equations is briefly described.

2.2.1. Rules of Equational Deduction

By 2.1.5, Given a signature, an equation of sort s over Σ is a pair $\langle t1,t2\rangle$ where t1 and t2 are both Σ-terms of sort s. We can also write $\langle t1,t2\rangle$ as form t1=t2. An equation t1=t2 is **satisfied** by a Σ-algebra A iff all the equations of the form $(\forall X)$ t1=t2 are satisfied by A, where X includes all variables occurring in t1 and t2. The following are the rules of equational deduction, given an equational presentation (Σ,E):

(1) *Reflexivity*. Each equation $(\forall X)t=t$ is derivable.

(2) *Symmetry*. If $(\forall X)t1=t2$ is derivable, then so is $(\forall X)t2=t1$.

(3) *Transitivity*. If the equations $(\forall X)t1=t2$, $(\forall X)t2=t3$ are derivable, then so is $(\forall X)t1=t3$.

(4) *Substitutivity*. If $(\forall X)t1=t2$ is derivable, then for any map $h:X\rightarrow T_\Sigma(Y)$, $(\forall Z)h^*(t1)=(\forall Z)h^*(t2)$ is derivable. where

$Z=X \cup Y$, h^* is the unique homomorphism: $T_\Sigma(X) \rightarrow T_\Sigma(Y)$.

(5) *Abstraction*. If $(\forall X)t1=t2$ is derivable, if y is a variable of sort s and y is not in X, then $(\forall X \cup \{y\})t1=t2$ is deriable.

(6) *Concretion*. If $(\forall X)t1=t2$ is derivable, if $x \in X_s$ does not appear in either t1 or t2, and $T_{\Sigma,s}$ is not empty, then $(\forall X-\{x\})t1=t2$ is also derivable.

2.2.2. Soundness and Completeness Theorems

This section gives the basic soundness and completeness theorems for the rules of equational deduction given in **2.2.1.** The proofs can be found in [EM 85].

Theorem 23: *Soundness*. Given a set E of Σ-equations, if an equation is deducible from E using rules the (1)-(6), then it is satisfied by every (Σ,E)-algebra.

Theorem 24: *Completeness*. Given a set E of Σ-equations, then every equation satisfied by all the (Σ,E)-algebras is derivable from E using rules (1) to (6) in **2.2.1.**

Soundness and completeness of a set of deduction rules together imply that, for the class of (Σ,E)-algebras, the model theoretic notion of an equation being satisfied by an (Σ,E)-algebra coincides with the proof theoretic notion of the equation being derivable from the given equations by the rules of equational deduction.

2.2.3. Term Rewriting

Term rewriting with equations is well-known from elementary algebra where arithmetic expression are simplified according to certain rules. These rules, if applied to an expression, yield another expression which is equivalent. It is shown ([EM 85]) that proving with equational deduction rules and term rewriting with equations are equally powerful techniques for deriving equations.

Given a Σ-signature, an equation $(\forall X)t1=t2$ such that each variable occurring in its left-hand side t1 also occurs in its right-hand side t2, can be used as a *rewrite rule* as follows: A term t can be rewritten

to a term t′ if t contains s subterm that is a substitution instance of the left-hand side t1 and t′ is the result of replacing that subterm by the corresponding substitution instance of the right-hand side t′. This is often indicated with the notation t→t′.

Rewriting gives a *unidirectional* version of equational deduction. Under mild conditions on a set E of Σ-equations, every term can be rewritten to a unique canonical form. This means that the initial (Σ,E)-algebra is computable, since we can decide the word problem by rewriting and then comparing canonical forms. In this way, rewrite rules provide an operational semantics for all computable algebras. The *evaluation* of an expression is its canonical form after rewriting, and equality of terms is decided by identity of their canonical forms. This point of view is the basis for OBJ3.

[GM 86] shows that if the rewrite rules satisfy two conditions then the word problem is decidable, and can be decided by rewriting. The following are the definitions of these two conditions:

Definition 25: Given an equational presentation (Σ,E), let $\rightarrow$ be the one step rewriting relation (among Σ-terms). Then a term t0 is a **normal form** relative to $\rightarrow$ if it cannot be further rewritten. The relation $\rightarrow$ is called **terminating** if there is no infinite sequence of rewritings: t0→t1→...tn→... .

Notice that if a system is terminating, then every term rewrites to a (not necessarily unique) normal form after a finite number of rewritings.

Definition 26: The $\rightarrow$ is called **Church-Rosser** if for each term t0 and each pair of rewritings $t0 \xrightarrow{*} t1$ and $t0 \xrightarrow{*} t2$ we have that t1 and t2 rewrite to a common term t3.

That is, the rewrite rules should satisfy the conditions of terminating and Church-Rosser.

2.3. Order-sorted Algebras and Rewriting

Order-sorted algebra is a generalization of many-sorted algebra.

This generalization supports a theory of abstract data types with multiple inheritance, subsort polymorphism, and an operational semantics by order-sorted rewrite rules. The essence of order-sorted algebra is a partial ordering $\leq$ on a set S of sorts; this **subsorts** relation imposes the restriction on an S-sorted algebra A that if $s \leq s'$ in S then $A_s \subseteq A_{s'}$.

The extension of MSA to OSA can be found in [GM 88]. It was shown that essentially all concepts and theorems of MSA can be generalized to OSA without complication. The important results of [GM 88] are those in the order-sorted equational deduction, including a completeness theorem and an initial algebra construction for conditional equations.

Chapter 3
Algebraic Specification Language OBJ3

In this chapter, we introduce the language features of algebraic specification language OBJ3. It provides flexible mechanisms of modularization and parameterization to maximize conceptual clarity, modifiability and reusability. OBJ3 was first implemented as OBJT in 1977 by Joseph Tardo based on the error algebras. OBJ2 was implemented during 1984-85 by Kokichi Futatsugi and Jean-Pierre Jouannaud following a design based on order-sorted algebra. OBJ2 reduces order-sorted rewriting to many-sorted. The current version of OBJ3, which is used in this project, was implemented in Kyoto Common Lisp. It uses a simpler, more efficient operational semantics that does term rewriting directly at the order-sorted level.

OBJ3 has four kinds of entity at its top level: objects, theories, views and reductions. Objects and theories are both **modules**, and can import other previously defined modules; because of such importation dependencies, an OBJ3 program is conceptually a *graph* of modules, rather than a sequence. Modules have **signatures** that introduce new sorts and new operations among both new and old sorts. An OBJ3 **object** gives executable code for the sorts and operations in its signature. An OBJ3 **theory** defines properties that may (or may not) be satisfied by an object. Both kinds of modules can be parameterized, and a parameterized module comes with one or more theories to define its interfaces. A **view** is a binding of the entities in a theory signature to entities in a module, and also an assertion that the module satisfies the properties stated in the theory. Thus, a view both indicates how to apply a parameterized module to an actual parameter, and asserts its semantic appropriateness. An OBJ3 **reduction** evaluates a given expression relative to a

given object by interpreting equations as rewrite rules.

In the following, we will introduce the structure of OBJ3 objects, parameterized modules and module hierarchies.

3.1. Objects

The most basic OBJ3 unit is the object, which encapsulates executable code. Syntactically, an object begins with the keyword obj and ends with endo, and has four main parts: (1) a header, containing its name, parameters, interface requirements; (2) a signature, declaring its new sorts, subsort relationships, and operations; (3) declaration of imported module list; and (4) a body, containing equations. The keywords **obj ... endo** delimits an object and indicate that *initial algebra semantics* is intended for it.

OBJ3 has many built-in objects such as NAT, INT for natural numbers and integers; QID for identifiers; and BOOL for truth values.

3.1.1. Sorts, subsorts and retracts

Sorts are similar to types in conventional strong typing programming languages. Sort declaration in OBJ3 has the following syntax:

sort (<SortName>)+ .

where (<SortName>)+ means one or more occurrences of <SortName>, and <SortName> can be an identifier.

Based on order-sorted algebras, OBJ3's flexible subsort mechanism provides operation overloading that enables a simple but powerful polymorphism, and multiple inheritance in the sense of object-oriented programming that permits one sort to be a subsort of two (or more) others, each having various defined operations; then all these operations are inherited by the subsort. In addition, with the subsort mechanism, the difficulties for abstract data types, which are based on many-sorted algebras, with operations that are "partial" (such as tail for lists and push for bounded stacks) disappear by viewing the operations as total on the right subsorts (see the example in 3.1.5).

The basic form of a subsort declaration in OBJ3 is

subsort <SortName1> < <SortName2>

meaning that the set of things of <SortName1> is a subset (not necessarily proper) of the things of <SortName2>. The form

subsorts <SortList1> < <SortList2> < ... .

can also be used, meaning that each sort in <SortList1> is subsort of each sort in <SortList2>, and so on; the elements of the various lists must be separated by blanks. OBJ3 checks for cycles of subsorts, and complains if it finds any.

Based on subsorts mechanism, OBJ3 provides *retracts* to combine the flexibility of untyped languages with the benefits of strong typing. In a strong typed language, certain expressions may fail to typecheck at compile time, although intuitively they have a meaningful value. For example, if the factorial function is only defined for natural numbers, then, strictly speaking, the expression ((-6)/(-2))! is not well-formed, since the argument of the factorial function is a rational number. However, since it might actually evaluate to a natural number, OBJ3 gives such an expression the "benefit of the doubt" at run-time through retracts to lower the sort of a subexpression to the required subsorts. In the above expression, the parser inserts the retract

$r_{\text{Rat>Nat}}$: Rat -> Nat

to fill the gap, yielding the expression ($r_{\text{Rat>Nat}}$((-6)/(-2))!. Retracts disappear only if their arguments have the required sorts.

3.1.2. Operations

OBJ3 lets users define any syntax they like for operations, including prefix, infix, or more generally, **mixfix**, to make it maximally appropriate for any given problem domain. Obviously, there are many opportunities for ambiguity in parsing such a syntax. OBJ3's convention is that an expression is well-formed if and only if it has exactly one parse. An integer precedence attribute can be given to eliminate the ambiguity in the parsing (see 3.1.3 below).

There are two forms for declaring an operation. The first is the usual functional form of parenthesized prefix with commas. The general

syntax for such declaration is

```
op <Op-Id> : <SortList> -> <Sort> .
```

For example,

```
op f : S1 S2 -> S3 .
```

indicates f(X,Y) of sort S3 for sort X of sort S1 and Y of sort S2. Commas are required as separators in well-formed expressions using this syntactic form.

The second case, mixfix form, uses place-holders indicated by an underbar character. The syntax for mixfix operation declarations is

```
op <form> : <SortList> -> <Sort> .
```

where <form> is a non-empty string of characters containing exactly as many underbars as there are sorts in <SortList>. This form can be used in prefix, infix and outfix declarations as well. For example

```
op top_ : Stack -> Nat .
```

is a prefix declaration, and

```
op {_} : Int -> Set .
op _+_ : Nat Nat -> Nat .
```

are outfix, and infix declarations respectively. A mixfix declaration for conditional is

```
op if_then_else_fi : Bool Nat Nat -> Nat .
```

In fact, OBJ3 provides such a built-in conditional operation for each sort, so that users do not have to define it themselves.

Sorts, subsort relationship, and operations combined give a signature of the underlying order-sorted algebra for the object. So between the : and the -> in an operation declaration comes the arity of the operation, and after the -> comes its value sort. Constant declarations have no underbars and have empty arity.

3.1.3. Operator attributes

OBJ3 allows users to specify certain properties of an operation as attributes at the time of its syntax declaration. These properties include axioms like associativity, commutativity, and identity that have both syntactic and semantic consequences, as well as the others that

affect order of evaluation (E-strategy), and parsing (precedence). Such attributes are given in square brackets after the syntax declaration:

op <form> : <SortList> -> <Sort> [(<attri>)+] .

where <attri> can be **assoc**, **comm**, **id:** <op-id> where <op-id> is a constant operator, **memo**, **prec** n where n is an integer, and **strat** followed by a sequence of natural numbers.

For example, in

op _or_ : Bool Bool -> Bool **[assoc id: false]** .

assoc indicates that or is an associative binary infix operation on boolean values. This means that the parser does not require full parenthesization. It also gives the semantic effect of an associativity axiom. The attribute **id: false** gives the effects of the identity equations (B or false = B) and (false or B = B).

The attribute **comm** has the expected effect.

An integer precedence attribute can be given for parsing; the lower the integer, the higher binding the operation. For example, the built-in object **INT** might have

```
op _+_ : Int Int -> Int [assoc prec 8] .
op _*_ : Int Int -> Int [assoc prec 5] .
```

so that the expression A + B * C is parsed as expected A+(B*C).

Given an operation, the attribute **memo** causes the results of evaluating any term headed by this operation to be saved; thus the work of reduction is not repeated if that term appears again. OBJ3 uses hashing to implement this efficiently.

In general, a large parsed tree of expression will have different sites where rewrite rules might apply, and the choice of which rules to try at which sites can strongly affect both efficiency and termination. Each of OBJ3's operations can have its own evaluation strategy. An **E-strategy** is a sequence of natural numbers given as an operation attribute to help determine where and in what order to apply rules. For example, **if_then_else_fi** has the strategy (1 0), which says evaluate the first argument until it is reduced, then apply rules at the top (indicated by 0); whereas _+_ (on Ints) might have strategy (1 2 0), which

says evaluate both arguments before attempting to add them. The keyword strat is used in the attribute list for user defined E-strategies, as in

```
op _+_ : Int Int -> Int [assoc id : 0 comm strat (1 2 0) ] .
```

Default E-strategies are determined by looking at the rules for an operation to see which arguments have non-variables terms; those are the arguments that must be evaluated before rules are applied at the top.

3.1.4. Equations

So far we have considered mostly syntax. In addition to the mathematical semantics that is based on order-sorted algebra, OBJ3 has an operational semantics based on order-sorted rewriting. The semantics of an object is determined by its equations. Equations are written declaratively and interpreted operationally as rewrite rules, which replace substitution instances of lefthand sides by the corresponding substitution instances of righthand sides.

The basic syntax for an equation in OBJ3 is

```
eq : <exp1> = <exp2> .
```

where both <exp1> and <exp2> are well-formed OBJ3 expressions. There are also conditional equations, with syntax

```
ceq : <exp1> = <exp2> if <bexp> .
```

where <bexp> is an expression of sort Bool. The built-in object BOOL is implicitly imported into every module.

All the above expressions can use variables that have been previously declared with the syntax

```
var <var-name-list> : <sort> .
```

where the variable names in the var name list are separated by blanks. For example,

```
var I J K : Nat .
```

3.1.5. An example

Finally we conclude section 3.1. with an OBJ3 specification of integer list.

```
obj INT-LIST is sort List NeList .
    protecting INT .
```

```
    subsorts Int < NeList < List .
    op nil : -> List .
    op __ : List List -> List [assoc id: nil prec 5] .
    op __ : NeList List -> NeList [assoc prec 5] .
    op head_ : NeList -> Int [prec 6] .
    op tail_ : NeList -> List [prec 6] .
    var N : Int .
    var L : List .
    eq : head N L = N .
    eq : tail N L = L .
endo
```

The module importation declaration **protecting INT** will be discussed in 3.3. For now, we just claim that it will import the sort of the built-in object INT, **Int**, into the object INT-LIST. The specification introduces a subsort NeList of nonempty lists to make the (traditional partial) head and tail operations total. The precedence attributes of the operations help to reduce the use of parentheses, so that **head N L** will be parsed as head(N L) as expected.

Given the above specification, we can let OBJ3 to evaluate expressions for us. For example:

```
OBJ3> reduce in INT-LIST as :
    tail 2 nil 3 nil 4 nil .
reducing term: (tail (2 (nil (3 (4 nil))))
reduction result NeList: (3 4)
```

The characters in bold San Serif are input by user.

3.2. Parameterized modules

In OBJ3, there are two kinds of module: objects that encapsulate executable code, and define abstract data types; and theories that specify both syntactic structure and semantic properties of modules. Each kind of module can be parameterized, where actual parameters are modules too. Interfaces of parameterized modules are defined by theories, thus include semantic as well as syntactic constraints. For parameter instantiation, a **view** binds the formal entities in an interface theory to actual entities in a module, and also asserts satisfaction of the theory by the module. Parameterized modules maximize reusability by permitting "tuning" to fit a variety of applications.

3.2.1. Theories and parameterization

A theory defines the interface of a parameterized module, i.e., the structure and the properties required of an actual parameter for meaningful instantiation. In general, OBJ3 theories have the same structure as objects. The difference is that objects are executable, while theories just define properties. Semantically, a theory has a variety of models, all the order-sorted algebras that satisfy it, whereas an object has just one model (up to isomorphism), its initial algebra.

We give some examples here. First, the built-in requirement theory for an interface that only requires designating a sort from an actual parameter:

```
th TRIV is
     sort Elt .
endth
```

Next, the theory of total ordered sets, which are like partially ordered sets but for every pair of elements in the set the relation holds in one way to other. Its models have a binary infix Bool-valued operation < that is reflexive and transitive.

```
th TOTORD is
     sort Elt .
     protecting BOOL .
     op _<_ : Elt Elt -> Bool .
     var E1 E2 E3 : Elt .
     eq : E1 < E1 = false .
     eq : (E1 == E2) or (E1 < E2) or (E2 < E1) = true .
     ceq : E1 < E3 = true if E1 <= E2 and E2 <= E3 .
endth
```

A parameterized object may have one or more requirement theories; these are given in square brackets after object name. The requirement theories must have been defined earlier in the program. The following is a parameterized STACK object using the theory TRIV above.

```
obj STACK[X :: TRIV] is
     sort Stack NeStack .
     subsorts Elt < NeStack < Stack .
     op empty : -> Stack .
     op push : Elt Stack -> NeStack .
     op top_ : NeStack -> Elt .
```

```
    op pop_ : NeStack -> Stack .
    var X : Elt .
    var S : Stack .
    eq : top push(X,S) = X .
    eq : pop push(X,S) = S .
endo
```

3.2.2. Views and instantiation

A module can satisfy a theory in more than one way, and even if there is a unique way, it can be arbitrarily difficult to find. A view provides a notation for describing the particular ways that modules satisfy theories.

A view v from a theory T to a module M, indicated v: T=>M, consists of a mapping from the sorts of T to the sorts of M preserving the subsort relation, and a mapping from the operations of T to the operations of M preserving arity, value sort, and the attributes **assoc**, **comm**, and **id:**, such that every equation in T is true of every model of M. The syntax for view is as follows:

```
view <ViewName> of <ObjName> as <ThName> is
  (sort <Sort> to <Sort> .)*
  (var <VarList> : <Sort> .)*
  (op {<Sort>} : <OpExp> to {<Sort>} : <Term> .)*
endv
```

where (...)* means zero or more occurrences, and {...} means optional. We can define a view from TOTORD to INT as follows:

```
view VINT of INT as TOTORD is
  sort Elt to Int .
  var E E' : Elt .
  op Bool : E < E' to Bool : _>_ .
endv
```

which is a view using the > relation in INT.

Instantiating a parameterized object means providing actual objects satisfying each of its requirement theories. In OBJ3, the actual objects are provided through views. For example, if P[X :: TOTORD] is a parameterized object, then we can form

```
obj M is protecting P[VINT] . endo
```

OBJ3 also provides **default views** to avoid using explicit view

definitions whenever possible. We are not going to discuss the issue here, for details see [G 87]. As a special case, every module has a default view from TRIV using its primary sort as the target for Elt. So given STACK[X :: TRIV] defined above, we can directly write instantiated object STACK[INT], which specifies stack of integers.

3.3. Hierarchy of modules

OBJ3 modules can import other modules in three different ways, *using*, *protecting* and *extending*. These define three different restrictions on preserved properties of imported modules, and thus define three corresponding partial orders (i.e., hierarchies) among modules. The using hierarchy is the most general, and embeds the other two.

The syntax for importing modules is

```
protecting
extending     <ModuleList>
using
```

where <ModuleList> is a list of module expressions, in particular, it can be a list of module names.

The meaning of these three import modes is related to the initial algebra semantics of objects, in that an importation of module M′ by M is:

1. protecting iff M adds no new data items of sorts in M′, and also identifies no old data items of sorts in M′;
2. extending iff M identifies no old data items of sorts in M′; and
3. using if there are no guarantees at all.

"Protecting" is the most restrictive relation, indicating that both the "no confusion" and "no junk" properties are preserved, and thus the imported module remains unchanged, hence the code can be shared. It has the advantage that it guarantees the E-strategies of imported operations do not need to be recomputed. "Extending" is an easy-to-check sufficient condition for "no confusion", requiring that the operations defined in an imported module do not occur as topmost symbols on the lefthand side of a new equation. For an extending importation, the E-strategies associated to the imported operations may have to be recomputed (see

[G 87]). "Using" is implemented by copying the imported module's text, without copying the modules that it imports; if desired, these can also be copied, just by listing them in the **using** <ModuleList> as well. For "protecting" and "extending", if a module M imports a module M' that imports a module M'', then M'' is also imported into M.

The **renaming** mechanism in OBJ3 allows one to rename the sorts and operations in the imported module by using sort mapping and operation mapping. For example,

```
obj INTSTACK is
     protecting STACK[INT] * (sort Stack to IntStack) .
endo
```

Chapter 4
Algebraic Denotational Specifications Using OBJ3

Since its invention, denotational semantics has become a powerful tool for the design and development of programming languages. Now more and more languages are given denotational semantics and studied and implemented based on their denotational definitions. Its popularity also gives rise to the necessity for structuring semantic definitions. It has been realized that even fairly simple definitions are usually wrong as first written and need to be debugged in the same way that programs are; this is especially true for the students who are just beginning to learn denotational semantics. In this and next chapters, we are going to study the method of using algebraic specification language OBJ3 to give denotational definitions for the programming languages. Since OBJ3 provides modularization and parameterization mechanisms for the semantic definitions, the resulting definitions are more readable and comprehensible; and since our definitions can actually be executed, the resulting definitions are more trustworthy.

Our semantic definitions are denotational, in the sense that meanings are always elements of particular abstract data types, i.e. the initial algebra of OBJ3 specification of a semantic domain. Our specification are also compositional, in the sense that the meaning of each syntactic phrase is composed from the meaning of its component phrases by the abstract syntax. The proposed structure of semantic definitions in OBJ3 follows closely to that of standard denotational semantics (of [Sc 86]). Hence we can easily test the correctness of the original denotational definitions, or write denotational definitions directly in OBJ3 language.

The major difference between our (algebraic) denotational defini-

tions and standard denotational semantics is that our specifications of semantic definitions are first-order in nature. Because OBJ3 is a first-order algebraic specification language, we have to specify every syntactic domains, semantic domains and valuation functions as first-order objects. Denotational semantics use *curried* operations widely in the semantics algebras and valuation functions. In our specifications we have to decurry those operations; 4.2.2 explains this in detail. The other major difference is that we adopt sets (or predomains) as the semantic domains. It is necessary because in our OBJ3 specifications, we are actually using the initial algebra of domain specification as semantic domains for the languages.

In this chapter we will first briefly introduce some basic concepts of denotational semantics. The accounts are mainly based on the book [Sc 86]. Following is the presentation of the basic structure of our algebraic denotational specifications. We will discuss the specification of syntactic domains from the abstract syntax. The problem of semantic domain specification is left for the next chapter, which will also describe three complete language definitions included in the appendixes.

4.1. An Introduction to Denotational Semantics

In the early 1970's, Scott and Strachey developed a mathematical approach to the programming language semantics ([Sc 86], [St 77]). In their approach, a language is given semantics by mapping each of its syntactic constructs into a set of mathematical domains, called domains of denotation. The success of this approach depends on how nice and convenient the mathematical properties of these domains are, and whether these domains are powerful and general enough to be used in giving semantics to a large class of programming languages. In fact, a number of existing languages, such as ALGOL60, Pascal, and LISP, have been given denotational semantics. This approach has also been used to help design and implement languages such as Ada, CHILL, and Lucid.

The denotational semantics are compositional semantics in the sense the meaning of a phrase is determined by the meaning of its constituent

subphrases. Typically, a denotational definition consists of three parts: abstract syntax of the language, semantic domains (value domains), and valuation functions that map the syntactic categories into semantic domains. In the following, we introduce some basic concepts with respect to these three components.

4.1.1. Abstract Syntax

Syntax of programming languages is usually given in BNF form. A formal description of the syntax involves a precise specification of the alphabet of allowable symbols and a finite set rules specifying how symbols may be grouped into expressions, commands, and programs. There are two kinds of syntax: one to determine the derivation of a phrase, called *concrete syntax*, and one to determine the semantics of a phrase, called *abstract syntax*. In denotation semantics, we assume that phrases are represented as a derivation tree after parsing. Hence we are dealing with abstract syntax, the ambiguity in grammar does not concern us.

Abstract syntax describes structure of the language. Set theory gives an abstract view of abstract syntax. Each nonterminal in a BNF definition names the set of those phrases that have the structure specified by the nonterminal's BNF rule. In denotational semantics, the

Figure 4.1 Abstract Syntax for Language BLOK1

```
P ∈ Program
K ∈ Block
D ∈ Declaration
C ∈ Command
E ∈ Expression
B ∈ Bool-Expression
I ∈ Identifier
N ∈ Numerals

P ::= begin K end
K ::= let D in C
D ::= D1 ; D2 | Const I N | Var I
C ::= C1 ; C2 | I := E | while B do C | if B then C1 else C2 | K
E ::= E1 + E2 | I | N
B ::= E1 eq E2 | not B
```

term *syntax domain* is used to stand for a collection of values with common syntactic structure; and a language's syntax is given by listing its syntax domains and its BNF rules. Figure 4.1 is an abstract syntax defining a block-structured language BLOK1, whose semantics is defined in Appendix A.

4.1.2. Semantic Domains

In denotational semantics, the sets that are used as values spaces are called *semantic domain*. Scott's domain theory provides least fixed point semantics to the recursive specification of domains and functions among domains ([Sc 86], [St 77]). In technical terms, Scott domains are *complete partially-ordered sets*. A partial order is a transitive, reflexive, and antisymmetric relation. A partially ordered set S is called *directed* if, for any x, y $\in$ S, there exists a z $\in$ S such that x $\sqsubseteq$ z and y $\sqsubseteq$ z, i.e., any two elements of S have an upper bound in S. A partially ordered set D is called a complete partial order (**cpo**) if it has a least element, which is called $\perp$, and any directed subset of D has a least upper bound (lub) in D.

Sets also make good domains (see ch.3 of [Sc 86]). A predomain is a cpos that may lack the least element $\perp$. Hence an ordinary countable set S may be regarded as a predomain. Since we can define the order of count as the partial order, and obviously S is directed for given any x,y $\in$ S, if y is counted after x, then y $\sqsubseteq$ x. As we shall see, sets work well in our algebraic denotational specifications of semantics.

Accompanying a domain is a set of operations. A domain plus its operations constitutes a *semantic algebra*.

A primitive domain is a set whose elements are atomic. The domain *Nat* defined as following is a commonly used primitive domain in denotational definitions.

Domain *Nat* = N
Operations
    zero, one, two, ... : *Nat*
    plus : *Nat Nat* $\rightarrow$ *Nat*

The operations *zero, one, two,...* are constants. Each of the members of *Nat* is named by a constant. The *plus* operation is the usual function. Another widely used primitive domain is truth values *Tr*.

As expected, there are domain building constructions for creating new domains from existing ones. Each domain builder carries with it a set of operation builders for assembling and disassembling elements of the compound domains. The *product* (×) construction takes two or more component domains and builds a domain of tuples from the components. The construction for unioning two or more domains into one or more domains into one domain is *disjoint union*. Given domains A and B, the disjoint union builds the domain A+B, a collection whose members are the elements of A and the elements of B, labeled to mark their origins. The last domain construction is the *function space builder* (→). For domains A and B, the function space builder → creates the domain A→B, a collection of functions from domain A to range B. See [Sc 86] for a detailed description of these compound domains.

4.1.3. Valuation Functions

The valuation function maps a language's abstract syntax structures to semantic domains. The domain of a valuation function is the set of derivation trees of a language. The valuation function is defined structurally. It determines the meaning of a derivation tree by determining the meanings of its subtrees and combining them into a meaning for the entire tree. The valuation function is actually a collection of functions, one for each syntactic domain. A valuation function D for syntactic domain D is listed as a set of equations, one per option in the corresponding BNF rule for D.

Figure 4.2 (from [Sc 86]) is the denotational definition of binary numerals. It gives a good illustration of the structure of denotational definitions.

4.2. Algebraic Denotational Specifications

The basic structure of our specifications follows closely to the

Figure 4.2: Denotational Definition of Binary Numerals

```
Abstract syntax:
    B ∈ Binary-numeral
    D ∈ Binary-Digit

    B ::= BD | D
    D ::= 0 | 1

Semantic algebras:
I. Natural numbers
   Domain Nat = N
   Operations
    zero, one, two,... : Nat
    plus, times : Nat Nat → Nat

Valuation functions:
B : Binary-numeral → Nat
    B⟦BD⟧ = (B⟦B⟧ times two) plus D⟦D⟧
    B⟦D⟧ = D⟦D⟧

D : Binary-Digit → Nat
    D⟦0⟧ = zero
    D⟦1⟧ = one
```

structure of denotational definitions. Given a denotational definition, we first specify an object SYN that is a specification of syntactic domains from the abstract syntax portion of the definition; then we give specifications for each of the semantic domains used in the definition. Finally the valuation function is specified as the mapping among the domains given above within an object called VAL.

In the following, we explain the basic structure of our specifications. The problem of specifying semantic domains is only briefly discussed and the detailed accounts postponed to the next chapter. Sections 4.2.1, 4.2.3, and 4.2.4 present the basic components of the specification corresponding to those of denotational definition, and 4.2.2 discusses the transformation of *decurrying* that is used in the specifications of semantic algebra operations and valuation functions.

4.2.1. Abstract Syntax and Syntactic Domains

Our approach of writing specifications for syntactic domains from given abstract syntax coincides with that proposed in [GTWW 77], which was introduced as an application of initial algebra semantics to the semantics of abstract syntax. [GTWW 77] noted that initial algebra semantics formalized the abstract syntax by characterizing (up to isomorphism) the algebra of parse trees of a context-free grammar as the initial algebra over a certain signature corresponding to the grammar. Here we are only interested in writing correct algebraic specification (in OBJ3) from given abstract syntax, i.e. the initial algebra of the specification that reflects the syntactic domain of the language.

In the following, we illustrate the process of deriving OBJ3 objects of syntactic domains from the abstract syntax. [GTWW 77] and [GM 86] described the general process of constructing a signature $\Sigma(G)$ from the grammar G.

Let $G=\langle N,T,P\rangle$ be any context free grammar, where N is a set of nonterminals, T is a set of terminals, and $P \subseteq N \times (N \cup T)^*$ is a set of productions. We specify an OBJ3 object G with sort set N and operations among N as follows, for each $p \in P$:

(1) if p is of form A ::= B, where A, B $\in$ N, then

```
subsorts B < A .
```

is declared within obj G, meaning the syntactic domain of B is contained in that of A.

(2) if p is of form A ::= t, where A $\in$ N, t $\in$ T, then

```
op t : -> A .
```

is declared in G, meaning t is a constant of sort A.

(3) if p is of form A ::= S0 A1 S1 ... An Sn, where Si $\in$ T* and Ai $\in$ N, then a mix-fixed operator is declared

```
op S0_S1_..._Sn : A1 ... An -> A .
```

Usually, no equations will be given within object G. In case (3) the attribute prec (precedence of the operator) may be given to help parse the expression appropriately.

We illustrate the procedure with an example. Figure 4.1 is an abstract syntax for the language BLOK1. Note that identifiers and numerals are left unspecified. A corresponding OBJ3 specification of the syntactic domain is given in Figure 4.3.

In the object SYN-DOM, the built-in object NAT and QID are imported through "protecting" to stand for the syntactic domains Numerals and Identifiers. QID provides identifiers with the operations of equality and lexicographic order built-in, and QID identifiers begin with the apostrophe symbol, e.g., 'a, 'b, '100, etc. The object introduces sorts corresponding to all the syntactic domains listed in the grammar other than Numerals and Identifiers: Prog, Decl, Com, Expr, and Bexpr, further, imported sorts Nat and Id are declared as subsorts of Expr since they are part of the expressions, and Block is declared as subsort of Com for the similar reason.

Figure 4.3: OBJ3 Specification of BLOK1 Syntactic Domains

```
obj SYN-DOM is
  sorts Prog Block Decl Com Expr Bexpr .
  protecting NAT .
  protecting QID .
  subsorts Nat < Expr .
  subsorts Id < Expr .
  subsorts Block < Com .

  op begin_end : Block -> Prog [prec 9] .

  op _;_ : Decl Decl -> Decl [assoc prec 6] .
  op Var_ : Id -> Decl [prec 5] .
  op Const__ : Id Nat -> Decl [prec 5] .

  op _;_ : Com Com -> Com [assoc prec 6] .
  op _:=_ : Id Expr -> Com [prec 5] .
  op while_do_ : Bexpr Com -> Com [prec 5] .
  op if_then_else_ : Bexpr Com Com -> Com [prec 5] .

  op _+_ : Expr Expr -> Expr [prec 3] .
  op _eq_ : Expr Expr -> Bexpr [prec 4] .
  op not_ : Bexpr -> Bexpr [prec 4] .
endo
```

The use of precedence attribute deserves more explanations. As we can see from the example, OBJ3's flexible mix-fixed operation declarations give us natural and readable form of specification directly from the grammar. But due to the ambiguity nature of the grammar appeared in the denotational definition, we often need to use meta-symbols "(" and ")" in the phrases to help OBJ3 parse correctly and successfully. Fortunately OBJ3's mechanism for assigning precedence to operator symbols reduce the use of parentheses to a minimum. The principle of assigning precedence attribute to operator symbols is as follows. For an operator p of rank A1 ... An -> A, for any Ai, if all operator symbols of sort Ai do not have A in their arities, then the precedence of p should be lower (hence the larger number assigned) than those of all these operator symbols. Therefore, E_1 eq E_2 has lower precedence than E_1 + E_2 so that "E_1 + E_2 eq E_3" will be parsed as "(E_1 + E_2) eq E_3" as expected without the use of parentheses. For the same reason, "_:=_" has lower precedence than "_+_", and "let_in_end" has lower precedence than all the operator symbols of sorts Decl and Com.

The attribute assoc is also used in the composition operators "_;_" of both ranks "Decl Decl -> Decl" and "Com Com -> Com" to reduce the use of parentheses and enhance the readability. As another principle, the precedence of composition operator of sort A is usually lower than those of other operators of the same sort A. For example, "_;_" of sort Decl has the lower precedence than those of "Var_" and "Const__" so that

```
Var 'x ; Var 'y ; Const 'n 2
```

will be parsed as

```
((Var 'x) ; ((Var 'y) ; (Const 'n 2))).
```

With the specification shown in Figure 4.4, only "If_then_else_" or "while_do_" may need parentheses to group the commands of "then" and "else" or "do" components. For example, the following piece of program will get parsed successfully

```
begin
    let
        Var 'x ; Var 'y ; Const 'n 10
```

```
    in
        'x := 'n ;
        if 'x eq 7 then ('y := 1 ; 'x := 'x + 'y)
        else if not ('x eq 3) then 'y := 2 else 'y := 3 ;
        'y := 'x + 'n
end
```

4.2.2. Decurrying and Lambda-lifting

In denotational semantics, curried operations are used extensively to define operations of semantic algebras and valuation functions. A curried operation f has the functionality $A_1 \rightarrow A_2 \rightarrow \ldots \rightarrow A_n \rightarrow B$ that take arguments one at a time. In our first-order algebraic framework, however, we have to *decurry* curried operation f to its original form: f': $(A_1 \times A_2 \times \ldots \times A_n) \rightarrow B$. Since all domain A_i's are to be specified by first-order objects in OBJ3, and in denotational semantics f will ultimately be used in a nested combination $((\ldots((f\ E_1)E_2)\ldots)E_n)$, the converted f' is isomorphic to f (See [Sc 86], [Sc 86a]).

The decurrying transformation has a very close relation to a functional language compilation strategy called the *lambda-lifting* [PJs 87], which is aimed at transforming a lambda expression into a form in which the lambda abstractions are particularly easy to instantiate. These special lambda abstractions are called *supercombinators*. Following [PJs 87], a supercombinator, $S, of arity n is a lambda expression of the form: $\lambda x_1.\lambda x_2 \ldots \lambda x_n.E$ where E is not a lambda abstraction (this just ensures that all the leading lambdas are accounted for by $x_1..x_n$) such that

(1) $S has no free variables,

(2) any lambda abstraction in E is a supercombinator,

(3) and $n \geq 0$, i.e., there may be no lambdas at all.

A supercombinator definition is of form: $S x_1 ... x_n = E. Thus a supercombinator based compilation strategy will regard the supercombinator definitions as a set of rewrite rules. A crucial point is that a supercombinator reduction only takes place when all the arguments are present. If we regard a supercombinator as a curried operation, the supercombinator definition and its use as rewrite rules act just like a

corresponding decurried operation.

From now on, we will specify any operations appeared in denotational definitions in their decurried form without notice.

4.2.3. Sets and Semantic Algebras

As discussed in 4.1.2, countable sets (just called sets hereafter) can be regarded as predomains, i.e., the Scott domain (cpos) that may lack the bottom element. We claim that in our first-order algebraic denotational specifications, sets are appropriate as semantic domain, since the sequencing and termination questions are answered automatically by the operational semantics of our algebraic specification. The other use of bottom element to denote error can be achieved by incorporating the single element domain *Unit* (Ch.3 of [Sc 86]). We shall see that we can specify a large variety of denotational semantics using sets.

The most important results of Scott's domain theory ([St 77], [Sc 86]) are the least fixed point semantics of recursive functions among domains, and an inverse limit constructions of reflexive domains. Since we are in the framework of first-order algebraic specification, we can conveniently express fixed-point properties by equations, but we cannot specify least fixed points directly (see [BW 87] for a discussion of this issue). That means we cannot apply the least fixed point semantics directly in our algebraic specifications. All we can do is to express the fixed-point properties of the recursively defined functions by equations, and hope that when regarding these equations as rewriting rules, it will terminate in reduction. Hence given a recursively defined function among domains, we can write equations for the function when restricting to the corresponding predomains or sets. When the reduction of the function application fails to terminate, we can say it evaluates to the bottom element $\perp$.

The domain theory also provides a solution to the recursively defined domains (reflexive domains). For the similar reason we cannot apply this solution in our first-order algebraic specifications. Section

5.3 presents our solution to the recursively defined sets based on the algebraic method.

To be specific, we regard the initial algebras of OBJ3 specifications as semantic algebras in the semantic definitions. If elements of any sort in a algebra constitute a countable set, this algebra can be seen as a predomain. It is obvious that when limited to sets, compound domain constructions of disjoint union, product, and function space preserve sets, i.e., the results of constructions are also sets.

Having dealt with the above issues, we now concentrate on the problems of writing correct specifications for the semantic domains. A semantic domain (a set particularly) together with its operations constitutes a semantic algebra. Although OBJ3 provides natural way of writing specifications for semantic algebras, it does impose special difficulties because of its first-orderness. Most of the semantic domains used in the denotational semantics are higher order, such as *store*, *environment*, and *continuation*. Chapter 5 is devoted to the issues of writing first-order specifications in OBJ3 for semantic domains. In this subsection, we only briefly cover the specification of primitive domains.

Primitive domains directly correspond to OBJ3's object. OBJ3 provides such built-in obj's as NAT, INT, BOOL, QID (identifiers), etc. It is easy to define other primitive semantics algebras in OBJ3.

For example, in denotational definitions, the domain *Unit* that contains only one element is useful for theoretical reasons. It can be used as an alternative form of error value, which we will see later; it can also be used whenever an operation needs a dummy argument. Following is an OBJ3 specification of domain *Unit*, the initial algebra of which contains exactly one element.

```
obj UNIT is sort Unit .
    op {} : -> Unit .
endo
```

As another example, the primitive domain of computer store locations is fundamental to the semantics of programming languages. Although the

members of domain *Location* are often treated as numbers, they are different in notion. Figure 4.4 is an OBJ3 specification of the domain *Location*. Note that the operator L is used to distinguish the *Location* from natural numbers. In the specification, the built-in operator **s** of NAT is a successor operator.

4.2.4. Valuation Function Specifications

The specifications of semantic mappings, i.e., valuation functions, are done by importing syntactic domains and semantic domains involved within an valuation object and specifying the valuation functions as operations among the sorts of domains imported. Usually, a valuation function in denotational definitions is in "curried" form, i.e., of the form

$$f: S \rightarrow A1 \rightarrow \ldots \rightarrow An \rightarrow A$$

where S is a syntactic domain, each Ai and A are semantic domains. As discussed in 4.2.2, in our specifications we will decurry the valuation functions.

In the following, we give a specification (in Figure 4.5) for the denotational semantics of binary numerals given in Figure 4.3. The next chapter will explain the specifications of three denotational semantics in the appendixes.

In Figure 4.5, the object SYN-DOM of syntactical domain is given according to the method introduced in 4.2.1. The constant operators 0

Figure 4.4: Computer store location specification

```
obj LOCATION is sort Location .
    protecting NAT .
    op first-locn : -> Location .
    op next-locn_ : Location -> Location .

    op L_ : Nat -> Location .

    var N : Nat .
    eq : first-locn = L 0 .
    eq : next-locn (L N) = L (s N) .
endo
```

Figure 4.5: Specification for Binary Numerals

```
obj SYN-DOM is sort Digit Binary .
  subsorts Digit < Binary .
  op 0 : -> Digit .
  op I : -> Digit .
  op __ : Binary Digit -> Binary [assoc] .
endo

obj VAL is
  pr NAT .
  pr SYN-DOM .
  op B[_] : Binary -> Nat .
  op D[_] : Digit -> Nat .
  var B : Binary .
  var D : Digit .
  eq : B[ B D ] = (B[ B ] * 2) + D[ D ] .
  eq : B[ D ] = D[ D ] .
  eq : D[ 0 ] = 0 .
  eq : D[ I ] = 1 .
endo
```

and I of sort Digit stand for the binary digit 0 and 1 respectively. They are declared so as to distinguish them from the semantic 0 and 1 of sort Nat. Later in the three semantic definitions in appendixes, we no longer attempt to make such distinguishes. In the other words, we will use the built-in objects NAT and QID as both syntactical domains and semantic ones. The only semantic domain appeared in Figure 4.1 is the domain of natural numbers. In the object of valuation function, the object NAT is imported to play the role of semantic domain natural numbers. In the object VAL, the valuation functions of B and D are declared as operations of forms

```
op B[_] : Binary -> Nat .
op D[_] : Digit -> Nat .
```

which are in mixed forms. As we can see, with such declaration, the equations for B[_] and D[_] are almost in same form as those in the original denotational definition.

Chapter 5
Semantic Algebra Specifications

The semantic algebras play an important role in denotational semantics. In this chapter we will present our methods of specifying various semantic algebras in the first-order algebraic language OBJ3. As pointed out in the last chapter, in our specifications of denotational semantics, we will adopt sets (in many-sorted algebras) instead of cpos as semantic domains. This is appropriate for our framework since the initial (term) algebras of specifications are intended as semantic domains. Most of semantic domains appeared in denotational semantics are high order function domains, 5.1 will introduce two methods of converting them into first-order domains: defunctionalization and λ-calculus. In section 5.2, we will discuss the methods of specifying compound domains. Section 5.3 introduces the specification of recursively defined domains. We will present a solution to the recursive specification of domains based on algebraic methods. Finally in section 5.4, we will discuss the specification of a special kind high order domains: continuations.

Throughout this chapter, three complete specifications of denotational definitions contained in Appendixes will be used as illustrations of our methods. Appendix A contains the direct semantics and its OBJ3 specification for the block-structure language BLOK1. Appendix C gives a similar language BLOK2 a continuation semantics and corresponding specification. A specification of denotational semantics for an applicative language PLISP is also given in Appendix B. Our experiences mainly came from the efforts in giving OBJ3 specifications for these three moderate denotational definitions.

5.1. Function Domain Specifications

To specify function domains in first-order algebraic language OBJ3,

we have to convert them to first order domains. In this section, we are going to introduce two ways in which a function domain D = A $\rightarrow$ B can be converted to a first order domain. If function domain D has a finite domain A, we would represent the members of D as tuples called the *closure* of D. The process of conversion called *defunctionalization* ([Sc 86a], [Sc 86]). Otherwise, D has infinite domain A; we treat members of D as lambda expressions and define λ-calculus over D. We will show that by using normal order reduction, we can eliminate the needs for arbitrary renaming of variables in the substitution rules if the expression started contains no free variables. Thus it is possible to give complete specifications for the λ-calculus in first-order equational specification language OBJ3.

5.1.1. Defunctionalization

Given a function domain D = A $\rightarrow$ B, the abstraction, i.e., a D-

Figure 5.1: Parameterized Object for Defunctionalized Domain

```
th DOMAIN is
  sort DElt .
  pr BOOL .
  op  _eq_ : DElt DElt -> Bool .
endth

th RANGE is
  sort RElt .
  op ? : -> RElt .
endth

obj FUN [A :: DOMAIN, B :: RANGE] is
  sort Fun .
  op nullF : -> Fun .
  op [__]_ : DElt RElt Fun -> Fun [strat (3 2 1 0)] .
  op __ : Fun DElt -> RElt .

  var x x' : DElt .
  var y : RElt .
  var f : Fun .
  eq : nullF x' = ? .
  eq : ([ x y ] f) x' = if x eq x' then y else f x' fi .
endo
```

valued term, is represented by a list of tuples of form <a,b> where a ∈ A, b ∈ B. Figure 5.1 gives a parameterized specification of such defunctionalized domain.

In the object FUN, a function is represented as a list:

[a1,b1]...[an,bn] nullF

where ai is a member of domain A, bi a member of range B. nullF is a function that maps every member of A to a designated member ? in B; it is regarded as a start point to construct a function in the object FUN. Note that the members of sort Fun in the object FUN are complete functions in the sense they automatically map every member of A other than those explicitly defined in the abstraction to a particular member of B. The interface theory RANGE reflects the requirement of existence of such element in any domain that is to be used as a range (codomain) of a such function. The theory DOMAIN also requires a predicate on the equivalence of two members of domain A.

Figure 5.2: Semantic Algebra Store

```
Domain v ∈ Storable-Value = Nat + Uninitialized
  where Uninitialized = Unit .
Operations
  add : Storable-Value Storable-Value → Storable-Value
    add = λv1.λv2. cases(v1) of
        isNat(n1)→(cases(v2) of
             isNat(n2)→inNat(plus(n1,n2))
            []isUninitialized()→inUnintialized() end)
       []isUninitialized()→inUninitialized() end

Domain s ∈ Store = Location → Storable-Value
Operations
  newstore : Store
    newstore = λl.inUninitialized()

  access : Location → Store → Storable-Value
    access = λl.λs.s(l)

  update : Location → Storable-Value → Store → Store
    update = λl.λv.λs.[l→v]s
```

The semantic algebra *Store* (see Figure 5.2) in denotational semantics is often defined as a function from *Location* to a certain domain *Storable-Value*. In figure 5.2, the semantic domain *Storable-Value* is defined as a disjoint union of domain of natural numbers and a single element domain called *Uninitialized*. The purpose of defining *newstore* as a mapping from every *location* to the element of *Uninitialized* instead of zero in *Nat* is to capture more potential errors in the definition of language's semantics and errors in programs when using language's semantic definition to perform verification and correctness proofs. As the denotation for the operation *update*, function updating expression "$[l \rightarrow v]s$" stands for the store that acts like s except that it maps the specific value l to v.

Figure 5.3 is an OBJ3 specification of semantic algebra *Store* by instantiating parameterized object FUN to the actual objects LOCATION and STORABLE-VALUE. The object LOCATION was specified in the last chapter.

In the object of STORABLE-VALUE, by importing NAT and declaring sort Nat as a subsort of Storable-Value, and by declaring a constant operation "uninitialized" of sort Storable-Value, we actual get a specification whose initial algebra contains exactly the disjoint union of Nat and single element domain Unit. Section 5.2 will discuss in detail the specification of compound domain. We used an overloaded operator + to make our specification of the operator *add* much simpler. Since Nat is a subsort of Storable-Value, when the arguments of + are all of sort Nat, the expected operation plus (+) will be performed on them. Having defined the views of LOCATION and STORABLE-VALUE as satisfied parameters to the object FUN, the specification of object STORE is almost a direct translation from the denotation in Figure 5.2. except that the operations are defined as "decurried" version of those in the original denotational definition.

Note that the initial algebra of our specification of store is not isomorphic to the semantic algebra in Figure 5.2, since there are more

Figure 5.3: Store Specification

```
obj STORABLE-VALUE is sort Storable-Value .
  pr NAT .
  subsorts Nat < Storable-Value .
  op uninitialized : -> Storable-Value .

  op _+_ :
       Storable-Value Storable-Value -> Storable-Value [assoc comm] .

  var N : Storable-Value .
  eq : N + uninitialized = uninitialized .
endo

view VLOC of LOCATION as DOMAIN is
  sort DElt to Loc .
  var L L' : DElt .
  op Bool : L eq L' to Bool : L == L' .
endv

view VSTVALUE of STORABLE-VALUE as RANGE is
  sort RElt to Storable-Value .
  op RElt : ? to Storable-Value : uninitialized .
endv

obj STORE is
  pr FUN [VLOC, VSTVALUE]
    * (sort Fun to Store, op (nullF) to (newstore)) .
  op access__ : Loc Store -> Storable-Value .
  op update___ : Loc Storable-Value Store -> Store [strat (3 2 1 0)] .
  var L : Loc .
  var V : Storable-Value .
  var S : Store .
  eq : access L S = S L .
  eq : update L V S = [ L V ] S .
endo
```

than one list of tuples corresponding to one abstraction of store. But it is easy to show that every store that was representable using the store operations in Figure 5.2 is representable using the OBJ3 version of specification. Furthermore, any reduction using a higher-order store can be simulated by a reduction that uses the corresponding first-order store.

We will use the parameterized object FUN to specify another commonly used semantic algebra *Environment* in section 5.2.3.

5.1.2. Lambda Calculus

Quite often we have to define a function domain D:A→B that does not have a finite domain A. In this case, we may use some syntactic form to record the abstraction of the function and specify certain reduction rules when a value of B is needed in the function application. In this way, functions can be treated as "first-order" objects, therefore we can still use first-order algebraic specification to define such function domains. Lambda notations are used extensively in denotational semantics. In this subsection we attempt to specify the function domain by regarding λ-expression and its conversion rules as an implementation of functions. The main advantage of this approach is its direct translation of lambda expressions into our first-order algebraic specifications, although it is less efficient than defunctionalization presented above. In the following, we use lambda expressions with integer as an atomic domain to illustrate our method. Note that actually we are going to give a specification for the recusively defined domain E = Int + E → E. Section 5.3 will discuss the specification of recursively-defined domain in detail.

5.1.2.1. Lambda-expressions

The definition of λ-expressions can be expressed by the following syntax:

```
<expression> ::= <variable>
               ¦ <integer>
               ¦ <expression><expression>
               ¦ λ<variable>.<expression>
               ¦ <expression>+<expression>
               ¦ (<expression>)
```

Actually the primitive domain *Integer* can be any other domain, and the primitive operation "+" can be any operations on this domain; more than one operations can appear in the syntax. Here for illustration purposes we keep everything as simple as possible.

Following [St 77], we first introduce the notion of *free* and *bound* variables. A variable *x* is said to be *free* in an expression E if

(1) E is the variable *x*, (not if E is *y* and *y*<>*x*, or E is integer N); or

(2) E is application XY and *x* is free in X or Y; or

(3) E is abstraction λ*y*.X, *x*, *y* are different and *x* is free in X; or

(4) E is X+Y, and *x* is free in X or Y; or

(5) E is (X), and *x* is free in X.

Similarly, A variable *x* is said to be *bound* in an expression E if

(1) E is application XY, and *x* is bound in X or Y; or

(2) E is abstraction λ*y*.X, and *x*, *y* are same or *x* is bound in X; or

(3) E is X+Y, and *x* is bound in X or Y; or

(4) E is (X), and *x* is bound in X.

Note that a particular variable can occur bound at one place in an expression and free at another place. Moreover, a particular occurrence of a variable can be free in some subexpression, but bound in the overall expression.

5.1.2.2. Substitutions

Given a function in the form of λ-abstraction λ*x*.M, the evaluation of function application (λ*x*.M)N resorts to the substitution of the variable *x* with N in the abstraction body M. The following substitution rule ([St 77]) spells out in formal syntactic details exactly how to substitute N for *x* in M.

Let *x* be a variable and M and N expressions. Then [N/*x*]M is the expression M′ defined as follows:

(1) If M is a variable,

(a) if M is *x*, then M′=N;

(b) if M is not *x*, then M′=M.

(2) If M is an application XY, M′=([N/*x*]X)([N/*x*]Y).

(3) If M is X+Y, M′=([N/*x*]X)+([N/*x*]Y).

(4) If M is an abstraction λ*y*.X

(a) if $x=y$, then M′=M;

(b) if $x<>y$, and x is not free in X or y is not free in N, then M′ = $\lambda y.[N/x]X$;

(c) if $x<>y$, and x is free in X and y is free in N, then M′ = $\lambda z.[N/x]([z/y]X)$ where z is the variable that does not occur free in N or X.

The first three cases are straightforward. Case (4)(a) applies on encountering an abstraction whose bound variable is the same as that being replaced; the new binding takes precedence and shields the body X from the effects of the substitution. Case (4)(b) deals with those cases where there is no possibility of a name clash, either because the body X contains no free occurrence of the variable x (so no substitution will in fact be performed) or because the expression N (to be inserted) contains no free occurrences of y which would be caught by the bound variable y of the abstraction $\lambda y.[N/x]X$. If these conditions are not satisfied, then in case (4)(c) it is necessary to change the bound variable y to some other name which does not clash.

5.1.2.3. Conversions Rules

Having formally defined substitution, we can introduce the conversion rules for performing transformations on λ-expressions. We write X cnv Y to indicate that either side may be replaced by the other whenever one of them occurs as an expression or as a subexpression of a larger expression. The following are the three conversion rules.

α. If y is not free in M, then $\lambda x.X$ cnv $\lambda y.[y/x]X$.

β. $(\lambda x.M)N$ cnv $[N/x]M$

η. If x is not free in M, then $\lambda x.Mx$ cnv M.

We are interested in using these rules to evaluate λ-expressions, i.e., we try to eliminate as many abstractions as possible. alpha-conversion does not help us in this, but the other two rules, when used in the left-to-right direction, both replace an expression containing an abstraction with some other expression that is much simpler. For this reason this kind of conversion is called a *reduction* and the particular

expression which is replaced is called a *redex*. Hence an expression of the form $(\lambda x.M)N$ is called a beta-redex, and $\lambda x.(Mx)$ is called an eta-redex if x is not free in M. When indicating a reduction we often use the symbol red instead of cnv; A red B asserts that A may be transformed to B by one or more reduction steps. When an expression contains no more redexes, it is said to be in *normal form*. It is not always possible to reduce an expression to normal form, i.e. the reduction may never terminate. But if, in an effort to reduce an λ-expression, two different reduction sequences terminate, the **Church-Rosser Theorem** guarantees that the results will be the same (see chapter 5 of [St 77]). Thus no two orders of evaluation can give different normal forms, although some may fail to terminate.

5.1.2.4. Orders of Reduction

Two orders of reduction are often used in λ-calculus. In *normal order* evaluation, the leftmost redex is chosen to reduce at each stage. Thus no expression in the argument position of a beta-redex is evaluated until the redex has itself been reduced, which might eliminate the argument from the expression altogether. Moreover, normal order reduction is guaranteed to terminate with a normal form if any order of evaluation does. Another order of evaluation is *applicative order*. In this order, the operator and operand of an application (β-redex) are separately evaluated to normal form before the β-reduction is performed. Although applicative order is less powerful than normal order since it may fail to terminate while normal order can, it is often faster than normal order when it terminates. This is because applicative order evaluates operand only once before it is substituted into the body of the operator, whereas normal order evaluates them as many times as necessary after the substitution.

In the next subsection, we choose normal order reduction in our OBJ3 specification. The main reason is that by using normal order reduction, we can eliminate the need for case (4)(c) of the substitution rule if the expression started contains no free variables, thus we can give

first order specification of the substitution rule in OBJ3. Case (4)(c) of the substitution rule requires the naming of an arbitrary variable that would not occur free in an arbitrary expression. In the following we show that if the expression we start to reduce contains no free variables and normal order reduction is used, then at any stage of reduction case (4)(c) will never be applicable.

Now we show that for beta- and eta-reductions, if the redexes contain no free variables, then (1) the results of one step reduction contain no free variables either, and (2) for the β-reductions, the substitution involves no application of case(4)(c) in the substitution rule. For a beta-redex $(\lambda x.M)N$ containing no free variables, by the definition of occurrences of free variables, N has no free variables and M may contain free occurrences of the variable x but no other variables. Then the result of one step β-reduction $[N/x]M$ will also contain no free variables since the only possible occurrences of free variable x will be replaced by N which contains no free variables. For an eta-redex $\lambda x.(Mx)$ containing no free variables, M contains no other variables than x, moreover, M cannot contain the variable x since $\lambda x.(Mx)$ is an eta-redex. Therefore we prove the assertion (1). For (2), in the substitution $[N/x]M$ when $M=\lambda y.X$, since N contains no free variables, y will not occur free in N. Thus only case (4)(b) in the substitution rule applies.

Since normal order of evaluation always reduces leftmost (or outermost) redexs first, from the above accounts, it guarantees that if the expression we start to reduce contains no free variables, at each stage of the reduction the result will also contains no free variables. Therefore, the normal order of evaluation will involves no application of case (4)(c) in the substitution rule. Note that in denotational semantics, or even in λ-calculus, it is the usually situation that the expression at the beginning of the reduction contains no free variables. Hence our assumption is reasonable in this respect.

The eta-rule does not help us if we are only interested in getting primitive value whenever we can, and are not concerned about getting

Figure 5.4: Specification for Lambda Calculus

```
obj INT-LAMBDA is sort Exp .
  protecting INT .
  protecting QID .

  subsorts Int < Exp .
  subsorts Id < Exp .

  op _+_ : Exp Exp -> Exp .
  op &_._ : Id Exp -> Exp .
  op __ : Exp Exp -> Exp .

  op [_/_]_ : Exp Id Exp -> Exp .
  op red : Exp -> Exp .

  op isLambda : Exp -> Bool .

  var I I' : Id .
  var E E' E'' : Exp .
  var N : Int .

  eq : isLambda(I) = false .                                    --- e1
  eq : isLambda(N) = false .                                    --- e2
  eq : isLambda(& I . E) = true .                               --- e3
  eq : isLambda(E E') = false .                                 --- e4
  eq : isLambda(E + E') = false .                               --- e5

  ceq : red(E E') = red(red(E) E') if not isLambda(E) .         --- e6
  eq : red((& I . E) E') = red([ E' / I ] E) .                  --- e7
  eq : red(E + E') = red(E) + red(E') .                         --- e8
  eq : red(N) = N .                                             --- e9
  eq : red(& I . E) = (& I . E) .                               --- e10

  eq : [ E / I ] N = N .                                        --- e11
  eq : [ E / I ] I' = if I == I' then E else I' fi .            --- e12
  eq : [ E'' / I ] (E E') = ([ E'' / I ] E) ([ E'' / I ] E') .  --- e13
  eq : [ E'' / I ] (E + E')
       = ([ E'' / I ] E) + ([ E'' / I ] E') .                   --- e14
  eq : [ E' / I ] (& I' . E) = if I == I' then (& I' . E)
                               else (& I' . ([ E' / I ] E)) fi . --- e15
endo
```

simplified function expression. Therefore the eta-rule is normally not included in studies concerning normal order reduction. In the following we will ignore eta-rule in our specifications.

5.1.2.5. Specification

Now we are able to present OBJ3 specification for the λ-calculus. In Figure 5.4, object INT-LAMBDA specifies the set of all λ-expressions with integer as an atomic domain, from the syntax given in 5.1.2.1. The syntactic constructors (_+_), (&_._), and (__) come directly from the syntax (throughout this paper, & always reads λ). The built-in objects INT and QID are imported for the syntactical categories Int and Id. Note that in the original syntax, many meaningless expressions may be passed as syntactical correct, such as (2 3) by application, (λ*x*.(1+*x*))+1 by addition, etc. Since conversion rules only work on the semantic correct expressions, in the specification the reduction of meaningless expressions is left unspecified. In section 5.3, we will explain the specification of semantics for an applicative language PLISP in Appendix B, which is essentially a λ-language. We will see that with the help of semantic domain of environment, these semantically incorrect phrases could be identified.

In figure 5.4, the auxiliary predicate isLambda_, which is defined by equations e1-e5, asserts whether an expression is a λ-abstraction. It is used in the reduction of application E E′ to see if the expression is a β-redex. The equations e11-e15 specify the substitution operations defined in 5.1.2.2 without the case (4)(c). The operation red(E) defined by e7-e10 reduces E to its normal form using only beta-rules.

Following is a demonstration of OBJ3 reduction of a λ-expression given in [St 77]. It can show how our specification works. The expression in original λ-notation is

(λ*p*. (λ*q*.(λ*p*.*p* (*p* *q*)) (λ*r*.*p*+*r*)) (*p*+4)) 2

which was input to OBJ3 in the specification notation and parsed, and is reduced as follows. Some spaces between symbols are omitted but are required in OBJ3 expression, the expression in bold is the focus of next step of OBJ3 rewriting.

```
    red((& 'p.((& 'q.((& 'p.('p ('p 'q)))(& 'r.('p + 'r))))('p + 4))) 2)
==> red([2/'p]((& 'q.((&  'p.('p ('p 'q))) (& 'r.('p + 'r)))) ('p + 4)))
                                                  { by e7: β-reduction }
```

```
==> red(([2/'p](& 'q.((& 'p.('p ('p 'q))) (& 'r.('p + 'r)))))
                                                    ([2/'p]('p + 4)))
                                        { by e13: substitution rule (2) }
==> red((& 'q.([2/'p]((& 'p.('p ('p 'q))) (& 'r.('p + 'r)))))
                                                    ([2/'p]('p + 4)))
                                    { by e15: substitution rule (4)(b) }
==> red((& 'q.([2/'p](& 'p.('p ('p 'q))) ([2/'p](& 'r.('p + 'r)))))
                                                    ([2/'p]('p + 4)))
                                        { by e13: substitution rule (2) }
==> red((& 'q.((& 'p.('p ('p 'q))) ([2/'p](& 'r.('p + 'r)))))
                                                    ([2/'p]('p + 4)))
                                    { by e15: substitution rule (4)(a) }
==> red((& 'q.((& 'p.('p ('p 'q))) (& 'r.[2/'p]('p + 'r))))
                                                    ([2/'p]('p + 4)))
                                    { by e15: substitution rule (4)(b) }
==>* red((& 'q.((& 'p.('p ('p 'q))) (& 'r.(2 + 'r)))) 6)
                                                { by e14, e11, e12 }
==> red([6/'q]((& 'p.('p ('p 'q))) (& 'r.(2 + 'r))))
                                                { by e7: β-reduction }
==> red(([6/'q](& 'p.('p ('p 'q)))) ([6/'q](& 'r.(2 + 'r))))
                                        { by e13: substitution rule (2) }
==>* red((& 'p.[6/'q]('p ('p 'q))) (& 'r.[6/'q](2 + 'r)))
                                    { by e15: substitution rule (4)(b) }
==>* red((& 'p.('p ('p 6))) (& 'r.[6/'q](2 + 'r)))
                                { by e13: substitution rule (2), and e12 }
==>* red((& 'p.('p ('p 6))) (& 'r.(2 + 'r)))        { by e13, e11, e12 }
==> red([(& 'r.(2 + 'r))/'p]('p ('p 6)))          { by e7: β-reduction }
==> red(([(& 'r.(2 + 'r))/'p]'p) ([(& 'r.(2 + 'r))/'p]('p 6)))
                                        { by e13: substitution rule (2) }
==> red((& 'r.(2 + 'r)) ([(& 'r.(2 + 'r))/'p]('p 6)))
                                    { by e12: substitution rule (1)(a) }
==> red((& 'r.(2 + 'r)) (([(& 'r.(2+'r))/'p]'p) ([(& 'r.(2+'r))/'p]6)))
                                        { by e13: substitution rule (2) }
==>* red((& 'r.(2 + 'r)) ((& 'r.(2 + 'r)) 6))            { by e12, e11 }
==> red([((& 'r.(2 + 'r)) 6)/'r](2 + 'r))         { by e7: β-reduction }
==>* red(2 + ((& 'r.(2 + 'r)) 6))                   { by e14, e11, e12 }
==> red(2) + red((& 'r.(2 + 'r)) 6)                              { by e8 }
==>* 2 + 8 = 10                                    { by e9 and e7 etc }
```

The OBJ3 reduction of the above expression is in Figure 5.5, which contains OBJ3 reductions of three λ-expressions. The first example shows that normal order of reduction is more powerful than applicative order one since normal order of evaluation will evaluate the expression $(\lambda y.0)((\lambda x.x\ x)(\lambda x.x\ x))$ to 0 while applicative order reduction will never terminate. The third example shows the reduction of a high order function application.

Figure 5.5: Example Runs of Lambda Calculus

```
Welcome to OBJ3 Version .99
system built: (1988 3 15 15 5 55)
Copyright 1987 by the OBJ3 Group (KF, JAG, JPJ, JM, TW, CK, HK, AM)
OBJ3> in ilambda
=======================================
obj INT-LAMBDA
=======================================

OBJ3> reduce in INT-LAMBDA as :
  red((& 'y . 0) ((& 'x . ('x 'x)) (& 'x . ('x 'x))))
.
reducing term: red(((& 'y . 0) ((& 'x . ('x 'x)) (& 'x . ('x 'x)))))
reduction result Int: 0

OBJ3> reduce in INT-LAMBDA as :
  red((& 'p . ((& 'q . ((& 'p . ('p ('p 'q))) (& 'r . ('p + 'r))))
                       ('p + 4))) 2)
.
reducing term: red(((& 'p . ((& 'q . ((& 'p . ('p ('p 'q))) (& 'r  . ('p
+ 'r)))) ('p + 4))) 2))
reduction result Int: 10

OBJ3> reduce in INT-LAMBDA as :
  red(((& 'f . (& 'x . ('f 'x))) (& 'x . ('x + 1))) 2)
.
reducing term: red((((& 'f . (& 'x . ('f 'x))) (& 'x . ('x + 1))) 2))
reduction result Int: 3
```

5.2. Compound Domain Specifications

In the previous section we presented OBJ3 specifications for function domain $F = A \rightarrow B$. In this section, we cover the specifications for the domains constructed by *disjoint union* and *product*. Such compound domains are used extensively in the denotational semantics. Together with function space builder, they constitutes a powerful tool for constructing semantic domains.

5.2.1. Product

The product construction takes two or more component domains and builds a domain of tuples from the component domains. We consider the

OBJ3 specification of the product domain of A_1, A_2, ..., A_n:

$P = A_1 \times A_2 \times \ldots \times A_n$

Assume that A_1 has been specified by the object SPEC-A_1 whose primary sort is Ai, for each of i ∈ [1..n]. Then we can specify the product P in the object SPEC-P as follows:

```
obj SPEC-P is sort P .
    protecting SPEC-A1 .
    protecting SPEC-A2 .
    ... ...
    protecting SPEC-An .

    op <_,_,...,_> : A1 A2 ... An -> P .
    op fst : P -> A1 .
    op snd : P -> A2 .
    ... ...
    op last : P -> An .

    var X1 : A1 .
    var X2 : A2 .
    ... ...
    var Xn : An .
    eq : fst(< X1 , X2 , ... , Xn >) = X1 .
    eq : snd(< X1 , X2 , ... , Xn >) = X2 .
    ... ...
    eq : last(< X1 , X2 , ... , Xn >) = Xn .
endo
```

where <_,_...,_> is an assembly operation for P, and fst, snd, ..., last disassembly operations. In fact, OBJ3 provides built-in parameterized object TUPLE[X :: TRIV, Y :: TRIV] for the case of the product of two domains. In the subsection section 5.2.3. we will show an example of product domain specification.

5.2.2. Disjoint Union

The construction for unioning two or more domains into one domain is called *disjoint union* or *sum*. Given two domains A and B, A+B is a collection of elements of A and B, with labels to mark their origins if confusion arises. There are two assembly operations associated with disjoint union A + B:

in_A: $A \rightarrow A + B$, and in_B: $B \rightarrow A + B$

which take elements of A or B and label their origins. The corresponding

disassembly operation "cases" combines an operation on A with one on B to produce an operation on the disjoint union A+B. If d is a value from A+B and $f(x)=e_1$ and $g(y)=e_2$ are the definitions of $f:A\rightarrow C$ and $g:B\rightarrow C$, then:

(cases d of $is_A(x)\rightarrow e_1 [] is_B(y)\rightarrow e_2$ end)

represents a value in C. The "cases" operation checks the tag of its argument, removes it, and gives the argument to the proper operation. The assembly and disassembly operations above can be generalized to sums of an arbitrary number of domains.

Now we concentrate on the OBJ3 specifications of disjoint union of domains $A_1, A_2, \ldots, A_n$: $S = A_1 + A_2 + \ldots + A_n$. Again, assume that, for each $i \in [1..n]$, domain A_i has been specified by object SPEC-A_i whose primary sort is Ai. If all these objects are disjoint in sorts Ai ($i \in [1..n]$), then we can use OBJ3's subsorts mechanism to specify S:

```
obj SPEC-S is sort S .
    protecting SPEC-A1 .
    protecting SPEC-A2 .
    ... ...
    protecting SPEC-An .
    subsorts A1 < S .
    subsorts A2 < S .
    ... ...
    subsorts An < S .
    op isA1 : S -> Bool .
    op isA2 : S -> Bool .
    ... ...
    op isAn : S -> Bool .

    var X1 : A1 .
    var X2 : A2 .
    ... ...
    var Xn : An .
    eq : isA1(X1) = true .
    eq : isA1(X2) = false .
    ... ...
    eq : isA1(Xn) = false .

    ... ...
    eq : isAn(Xn) = true .
endo
```

Since all Ai's are declared as subsorts of sum S, we do not need assem-

bly operations in Ai: $A_1 \rightarrow S$ as in denotational definitions. All terms of sort Ai can safely appear in places where sort S is required. Because of the "high-orderness" of original disassembly operation "cases", we define a set of predicates "isAi" to help the specification of disassembly operations. For any semantic equation in denotational definitions of form:

$$E = \text{cases a of } isA_1(a1) \rightarrow e1(a1) \; [] \; isA2(a2) \rightarrow e2(a2) \; \ldots \; [] \; isAn(an) \rightarrow en(an) \text{ end}$$

we use a set of OBJ3 conditional equations to specify the semantic equation:

```
ceq : E' = e1'(a) if isA1(a) .
ceq : E' = e2'(a) if isA2(a) .
... ...
ceq : E' = en'(a) if isAn(a) .
```

where E' is a OBJ3 version of the term E, and ei' is the corresponding term ei for each i $\in$ [1..n]. Note that in the OBJ3 equations, the term a is used in places of ai's in the terms of ei's, which is made possible by the OBJ3 retracts that lower the sort of a to that of ai in the parsing. The selection of equation ceq : E' = ei'(a) if isAi(a) will ultimately make the retract $r_{S>Ai}(a)$ disappear because of the condition isAi(a).

Remember that OBJ3 parser only inserts retracts for the subexpression, i.e. when the term to be retracted is an argument to some operator. When the "cases" operation has a portion of form

```
cases a of ... isAi(ai)→ai ... end
```

we need to define an operation to explicitly coerce a from sort S to sort Ai:

```
op selecAi : S -> Ai .
... ...
var Xi : Ai .
eq : selecAi(Xi) = Xi .
```

Note that selecAi essentially performs the same function as retract.

Now, that portion of the "cases" operation can be specified as:

```
ceq : E' = selecAi(a) if isAi(a) .
```

Again, the selection of the conditional equation will guarantee that selecAi(a) will be reduced to a.

In the above, we assume that all SPEC-Ai's are disjoint (or different) with respect to their primary sorts Ai's. In denotational semantics it is usually the case, but not always so. For example, the domain Poststore used in direct semantics (see the example in next section) is defined as Store + ErrStore where Errstore = Store. Suppose in the sum $S = A_1 + A_2 \ldots + A_n$, A_j and A_k are the same domain A. Assume that domain A has been specified by object SPEC-A whose primary sort is A. Then we can modify the object SPEC-S above as follows:

```
obj SPEC-S is sorts S Aj Ak .
     ... ...
     protecting SPEC-A .
     ... ...
     subsorts Aj < S .
     subsorts Ak < S .
     ... ...
     op inAj : A -> Aj .
     op inAk : A -> Ak .
     ... ...
     var Xa : A .
     eq : isAj(inAj(Xa)) = true .
     eq : isAk(inAk(Xa)) = true .
     ... ...
endo
```

In the modified object, two sorts Aj and Ak were introduced with two tagging operators inAj and inAk to make sort A into these two sorts respectively.

Having described generally our methods of specifying compound domains, in next subsection, we present as an example a specification of direct (denotational) semantics for the block-structured language BLOK1.

5.2.3. A Specification of Direct Semantics

Appendix A contains a complete denotational semantics for BLOK1 and its OBJ3 specification. The denotational semantics follows closely in style and convention to those of [Sc 86]. It is called direct semantics

because the valuation functions map syntactic domain of command to the operations on the store domain. The basic structure of specification has been introduced in chapter 4. Here we only give some explanations for semantic algebra specifications. The method of specifying syntactic domain from abstract syntax was introduced in 4.2.1.

Among 11 semantic algebras listed in part I: denotational semantics, the first 3 were specified by the built-in objects BOOL, NAT, and QID. The specifications of (4) Storage Locations, (5) Storable values, and (6) Store were discussed in 4.2.3 and 5.1.1. Semantic algebra (6) Poststore is defined as the disjoint union of Store and ErrStore where ErrStore itself is the domain Store. Using the method presented in the last subsection, we first import object STORE, declaring Store as a subsort of PostStore. In this way, the operation *return*:*Store*→*Poststore* is not necessary in the object POSTSTORE because of the subsort relation. To avoid any confusion, the operation *signalerr*:*Store*→*ErrStore* is used as a tag marking a store as an errstore. In addition, predicates isStore and isErrSTore are defined in the object POSTSTORE to help the specification of disassembly operation "cases...". Take semantic mapping for the command composition as an example. In the denotational definition, the valuation function for command has the functionality:

C: Command → *Env* → *Store* → *Poststore*

and the command composition is defined by the following equation:

$$\begin{aligned} C[\![C_1\ ;\ C_2]\!] = \lambda e.\lambda s.\ &\text{cases } (C[\![C_1]\!]\ e\ s) \text{ of} \\ &\text{is}Store(s') \rightarrow C[\![C_2]\!]\ e\ s' \\ [\!]\ &\text{is}ErrStore(s') \rightarrow \text{in}ErrStore(s') \text{ end} \end{aligned}$$

In the object VAL-FUN that specifies the valuation functions, we have the corresponding decurried operation:

```
op C[_]__ : Com Env Store -> PostStore .
```

and have the following equation concerning command composition:

```
eq : C[ C1 ; C2 ] e s = if isStore(C[ C1 ] e s)
                        then C[ C2 ] e (C[ C1 ] e s)
                        else signalerr(s) fi .
```

Since Poststore is the sum of only two domains, we used if_then_else_fi in the equation above instead of two conditional equations as proposed.

Note that although C[_]__ requires a Store as the third argument, OBJ3 can still parse the "then" subexpression, which is of sort PostStore, because of the "retract" mechanism. The condition in the "if" part will guarantee that the inserted retract will disappear at the time of reduction.

Now let's look at the specification of the semantic algebra *Environment*. Semantic algebra (8) *Denotable values* again is a disjoint union of *Location*, *Nat*, and error domain *Undefined*, which denote the values an identifier can have. The specification of this domain in the object DENOTABLE-VAL demonstrate the method of specifying sum domain presented in the this section. The semantic algebra (9) *Environment* is specified by instantiating the parameterized object FUN with objects QID and DENOTABLE-VAL to define the mapping from identifiers to their denotable values. The sort Env is constructed by the product <_,_> of Idmap and location which denotes the next available storage location. Using the operations provided by the parameterized object FUN, the specifications of the operations *emptyenv*, *updateenv*, and *accessenv* are almost direct translations from the original denotational definitions. Note that the operation *reserve-locn* : $Env \rightarrow (Location \times Env)$ was split into two operations in the object ENV:

```
op reserve-locn : Env -> Env .
op get-locn : Env -> Loc .
```

to simplify the specification. Let's look at the specification of the following semantic equation in the denotational semantics:

$$D[\![\text{Var I}]\!] = \lambda e.\ \text{let } (l,\ e') = (\textit{reserve-locn}\ e) \text{ in } (\textit{updateenv}\ [\![I]\!]\ \text{in}\textit{Location}(l')\ e')$$

The split operations reserve-locn and get-locn in the object ENV make the above equation appear in the object VAL-FUN as follows:

```
eq : D[ Var I ] e = updateenv I (get-locn e) (reserve-locn e) .
```

Finally, note that in the specification, objects NAT and QID were both used as syntactic domains and their corresponding semantic domains.

5.3. Recursive Domain Specifications

In denotational definitions, recursively defined domains of the form

D=F(D) are often used in defining semantic domains. Recursively defined domains are also called reflexive domains. Recall that domains are cpos in the denotational semantics. It is shown ([Sc 86]) that for any recursive domain specification of the D=F(D), where F is an expression built with constructors +, $\rightarrow$, $\times$, and lifting $\perp$ such that F(E) is a cpos when E is, there is a domain D_∞ that is isomorphic to $F(D_\infty)$. D_∞ is the least such cpo that satisfies the specification. The way D_∞ is constructed is called the *inverse limit construction*.

Since in our specifications we use predomains (or countable sets) as semantics domains and our specifications of semantic domains are declarative instead of constructive, we use a set theoretic approach in writing specifications for the recursively defined domains. Given a recursive definition *D*=F(*D*), we write an object SPEC-D whose primary sort is D, such that the set of terms of sort D (in the initial term algebra) $T_{\Sigma,D}$ satisfies equations $T_{\Sigma,D}=F(T_{\Sigma,D})$.

For example, we can actually view object INT-LAMBDA in 5.1.2.5 as a specification of recursively defined domain $E = Int + E \rightarrow E$ in the sense that the initial term algebra T satisfies the equation:

$$T_{Exp} = T_{Int} \cup (T_{Exp} \rightarrow T_{Exp})$$

where T_{Exp} is the set of all terms of sort Exp, and T_{Int} is the set of all terms of sort Int. Obviously, T_{Exp} has infinite number of elements.

In our specifications, we regard sets as semantic domains, and correspondingly, domain constructors (+, $\times$, $\rightarrow$) as set operations. In case of domain disjoint union A+B, if A and B are syntactically distinguishable, then A+B is equal (isomorphic) to A $\cup$ B; if not, a tag may be attached to A or B to avoid confusions. 5.2.2 has already discussed this issue. In the following we assume A and B are different in disjoint union A + B, hence it has the same effect as A $\cup$ B. Let D be a recursively defined domain of form

$$(*)\ D = A_1 + \ldots A_n + F_1(D,A_{11},\ldots,A_{1i_1}) + \ldots + F_t(D,A_{t1},\ldots,A_{ti_t})$$

where A_j ($1 \le j \le n$) and A_{pq} ($1 \le p \le t$, $1 \le q \le i_p$) are previously defined domains, and F_k ($1 \le k \le t$) is an expression consisting of operations +, × and $\rightarrow$, and the outermost operator is either × or $\rightarrow$. Now we specify D in the OBJ3 object SPEC-D as follows (again assume the other domains are specified in corresponding objects):

```
obj SPEC-D is sort D .
    ... ...
    subsorts A1 < D .
    ... ...
    subsorts An < D .
    op F1 : D A11 ... A1i1 -> D .
    ... ...
    op Ft : D At1 ... Atit -> D .
    ... ...
endo
```

Here we specify F_k ($1 \le k \le t$) directly as domain D constructors, and it is possible to define it completely using equations for the most recursively defined domains encountered in denotational definitions.

Given the above specification, we have the following set equation:

$$T_D = T_{A_1} \cup \ldots \cup T_{A_n} \cup F_1(T_D, T_{A_{11}}, \ldots, T_{A_{1i_1}}) \cup \ldots \cup F_t(T_D, T_{A_{t1}}, \ldots, T_{A_{ti_t}})$$

where T_D, T_{A_k}, and $T_{A_{pq}}$ are the sets of terms of sorts D, A_k ($1 \le k \le n$) and A_{pq} ($1 \le p \le t$, $1 \le q \le i_p$) respectively in the initial term algebra of the specification. This equation shows that the object SPEC-D indeed specified domain D with respect to the recursively defined set equation (*). Note that if $n>0$, the set of terms of sort D will not be empty.

We have to admit that our set theoretic approach is limited in solving recursively defined domains. In the equation (*), if $n=0$ then our specification will result in degenerated initial algebra. In the other words, recursively defined domains such as $E = E \rightarrow E$ and $E = N \times E$, which have solutions in the Scott's domain theory, will not

have a solution in our specification. This is a problem that our underlying algebraic approach can not handle. Fortunately most denotational definitions do not use such recursively defined domains, nor do our specifications of three modest denotational semantics. In the following, we will not try to deal with this issue any further.

In the next subsection, we illustrate our method with a specification of denotational semantics for the applicative language PLISP.

5.3.1. Semantic Specification of an Applicative Language

In this subsection, we will give some explanations for the specification of PLISP appeared in Appendix B. The part I of the Appendix B, denotational semantics, is adapted from section 7.2 of [Sc 86], with the slight modifications of syntax and semantic domains. PLISP is similar to pure LISP. A program in the language is just an expression. An expression can be a LET definition; a LAMBDA form representing λ-abstraction; a function application; a list expression using CONS, HEAD, TAIL, or NIL; an identifier; or a numeral. The semantics of this language is expressed by the semantic domain *Denotable-value*, which is a recursively defined domain:

$$\begin{aligned} \textit{Denotable-Value} &= \textit{Function} + \textit{List} + \textit{Nat} + \textit{Error} \\ \textit{Function} &= \textit{Denotable-value} \rightarrow \textit{Denotable-value} \\ \textit{List} &= \textit{Nil} + \textit{Denotable-value} \times \textit{List} \end{aligned}$$

where *Error* = *Unit*, and *Nil* = *Unit*. The semantic mapping **E** determines the meaning of an expression with the aid of an environment.

Now let us first look at the specification of recursively defined semantic domain *Denotable-value*. In the object DENOT-VAL, we use λ-notations to represent functions. Therefore, similarly to the object INT-LAMBDA presented in 5.1.2.5, the substitution operation "[_/_]_" and reduction operator "red" are declared and specified. Note that for simplicity, we again omitted the eta-rule, which is of no use in getting an atomic answer value in the reduction of function application. We specified object VARS as variables in the λ-abstractions. A variable, i.e. the element of sort Vars, has the form "v *n*" where *n* is a natural number. Variables in the body of an λ-abstraction can be substituted by

any elements of sort Den. Therefore, we declare sort Vars as subsorts of Den. The predicate isVarterm is declared to assert the variable terms such as hd (tl *x*), *f x*, etc. The existence of variable terms in expressions causes some problem in the specifications of predicates isList and isFunc, since the sort of variable terms is dependent on the substitution of variables. We solve this problem by temporarily assert isList or isFunc as true for the variable term as desired. The real capture of error expressions is delayed to the time of substitution (see the equations for the substitution rules). The list operations hd_, tl_, and _;_ (representing cons) are defined within the object DENOT-VAL as usual. And the predicates isFunc, isList, and isErr are defined for the disassembly operation in the original denotational definitions. Differently from the specification in Appendix A, we specify single element domain *Error* (and *Unit*) directly by introducing constant operation Err (and Nil). It is easy to see that the two methods generate isomorphic term algebras.

The specification of domain *Env* is similar to the one in Appendix A, which was explained in 5.3.2. Since we have to generate variables of denotable values for the valuation function of syntactical expression LAMBDA I E, the sort Env in the specification is declared as tuple

Env = (Id -> Den) × Vars

where the second element of tuple is used to indicate to current available variable of form "v *n*". In addition to the operations emptyenv, accessenv, and updateenv, getvar and gonextvar are specified to fetch the current available variable from an environment and to obtain an environment with next available variable from a given environment. We will show the use of these two operations below.

The specifications of syntactical domain and valuation functions are also similar to the ones in Appendix A. Here we only look at the specification for the semantic equation:

E⟦LAMBDA I E⟧ = λ*e*.in*Function*(λ*d*.E⟦E⟧(*updateenv* ⟦I⟧ *d e*)).

which is specified in the object DENOT-VAL as follows:

```
E[ LAMBDA I E ] e
= (& (getvar e) . (E[ E ] (updateenv I (getvar e) (gonextvar e)))) .
```

To make the specifications more efficient, the applicative order evaluation is used in the valuation function specification. But our specification of λ-reduction in the object DENOT-VAL still assume that case (4)(c) of the substitution rule in 5.1.2.2 will not be applicable in any stage of a reduction. Fortunately the assumption is true in the specification since variables are generated uniquely, of form (v N). For example, the expression LAMBDA 'x (LAMBDA 'x 0) has the following evaluation:

```
    E[ LAMBDA 'x (LAMBDA 'x 0) ] emptyenv
==> (& (v 0) . (E[ LAMBDA 'x 0 ] (updateenv 'x (v 0) e0)))
          where e0 = gonextvar emptyenv = < ¦ , (v 1) >
==> (& (v 0) . (& (v 1) . 0))
```

We can see that the variables of denotable values are generated with increased subscripts. Therefore in a β-redex (& V . M) N, if N contains any free variable V', then to be any semantic correct expression's denotation, (& V . M) N must be a subexpression of

```
& V' . ... (& V . M) N ...
```

that is to say, V' would have lower subscript that V, V will never occur free in N. Hence we proved that case (4)(c) of substitution rule is still not applicable in our λ-reduction of denotable values.

5.4. Specifications of Continuation Domains

The semantic domain that models controls, i.e., the evaluation ordering of a program's constructs, is called a *continuation*. Continuations were first developed for modeling unrestricted branches ("gotos") in a general purpose language. In this section we will apply the concept of continuation to every levels of semantic definition and present a specification of continuation-based semantics for BLOK2 that is similar to BLOK1 defined in Appendix A. Again the complete denotational semantics and corresponding OBJ3 specification are included in the Appendix C.

5.4.1. Continuation-based Semantics

In continuation semantics, the meaning of a program construct is a function that accepts the state (store) existing prior to execution, plus an additional argument called the continuation, and produces the final output of the entire program. The continuation that is provided as an additional argument is a function from the state existing after program construct execution to the final program output, which gives the semantics of the "rest of the computation" to be performed if the current construct terminates normally. Thus a program construct with normal behavior will produce its output by applying the continuation to the state (store) following execution. But an abnormal one can produce the final output in some other manner, possibly by ignoring the continuation.

The language BLOK2 defined in Appendix C is similar to BLOK1 in Appendix A, but augmented with a FORTRAN-like **stop** command. The evaluation of a stop in a program causes a branch to the end of the program, cancelling the evaluation of all remaining commands. The semantic domain of Command Continuations *Ccont* is defined as $Store \rightarrow Answer$, where *Answer* in the definition is product of domain *Message* and *Store*. Actually the domain *Answer* can be any domain of stores, output buffers, messages, etc. The valuation function for the command thus has the functionality:

$$\text{Command} \rightarrow Env \rightarrow Ccont \rightarrow Store \rightarrow Answer$$

The additional argument *Ccont* holds the rest of the program to the current command being executed. Although the current command never bothers to examine the rest of the program, i.e., the continuation, it has the authority to discard the continuation, as command stop does:

$$C[\![\text{stop}]\!] = \lambda e.\lambda c.\lambda s.\, terminate\ stopped\ s.$$

The operation *terminate* simply tuple the message *stopped* and the store s to produce an answer. The continuation c was discarded.

In the denotational definition of Appendix C, the notion of continuation was extended to the other levels of semantics. If we regard

command continuation as control domain of runtime semantics, the other continuations defined (expression continuations, identifier continuations, and boolean expression continuations) were aimed at capturing the control facets of compiling time semantics. The major concern here is that the use of an identifier may cause some error when referenced in the expression, or in the left-hand side of an assignment command. The domain of expression continuations is defined as:

$$Econt = Storable\text{-}value \rightarrow Ccont$$

meaning that the rest of program is waiting for a value in addition to the state (store) to produce an answer. *Econt* is used in the valuation function for expressions. Similarly, domains of *Lcont* and *Bcont* are defined for the valuation functions of boolean-expressions and left-hand side identifiers. Now we look at the semantic equations for the assignment command:

$$C[\![I := E]\!] = \lambda e.\lambda c.L[\![I]\!]\ e\ (\lambda l.E[\![E]\!]\ e\ (\lambda v.\ updatestore\ l\ v\ c))$$

The valuation function L: Identifier $\rightarrow$ *Env* $\rightarrow$ *Lcont* $\rightarrow$ *Ccont* has the following equation:

$$\begin{aligned} L[\![I]\!] = \lambda e.\lambda\beta.&\text{cases } (accessenv\ [\![I]\!]\ e) \text{ of} \\ &\quad isLocation(l) \rightarrow \beta\ l \\ &\quad [\!]\ isNat(n) \rightarrow terminate\ id\text{-}use\text{-}err \\ &\quad [\!]\ isUndefined() \rightarrow terminate\ id\text{-}undefined \end{aligned}$$

The valuation function for the assignment first forms an identifier continuation $\lambda l.E[\![E]\!]\ e\ (\lambda v.updatestore\ l\ v\ c)$ and then hands it to the valuation function L, which examines the identifier I; if I was declared as a variable then it will give the corresponding location l_0 to the identifier continuation, otherwise L will discard the continuation and produce a final answer by the operation *terminate* with appropriate message. Let us assume that the assignment I := E is semantically correct, then we have the following evaluation:

$$\begin{aligned} & C[\![I := E]\!]\ e_0\ c_0 \\ \Longrightarrow\ & L[\![I]\!]\ e_0\ (\lambda l.E[\![E]\!]\ e_0\ (\lambda v.\ updatestore\ l\ v\ c_0)) \\ \Longrightarrow\ & (\lambda l.E[\![E]\!]\ e_0\ (\lambda v.\ updatestore\ l\ v\ c_0))\ l_0 \\ \Longrightarrow\ & E[\![E]\!]\ e_0\ (\lambda v.\ updatestore\ l_0\ v\ c_0) \\ \Longrightarrow\ & (\lambda v.\ updatestore\ l_0\ v\ c_0)\ v_0 \\ \Longrightarrow\ & updatestore\ l_0\ v_0\ c_0 \end{aligned}$$

==> $\lambda s.c_0$ (*update* l_0 v_0 s)

The operation *updatestore* takes location, storable value and command continuation as arguments and produce a command continuation as result.

Compared with the direct semantics in Appendix A, the continuation semantics tends to use higher order domains in the definition. *Ccont*, *Econt*, *Lcont*, and *Bcont* are all higher order domains. In the next subsection, we discuss the specification of continuation semantics.

5.4.2. Specifications

Since continuations are all higher order domains, we could use λ-expressions and their conversion rules as implementation, just as we did in the section 5.1.2. But the continuation domains have their own features. They, as arguments, are just "place holders" in the sense that they just represent the (nominal) notion of "rest of computation", the actual evaluation of the "rest of computation" takes place when the required stores or values or locations are handed to them. For the domains of *Econt*, *Lcont*, and *Bcont*, which map some value domains to *Ccont*, the applications of continuations to the wanted values take place immediately, before further evaluation of the rest of the computation. The implications of these observations are two fold: first, complete specifications of continuations as λ-notations are becoming very complicated since the continuations are not in certain set of reduced forms and they involve the structure of syntactical domains; secondly, the partial specifications of continuations are possible because the number of nesting of λ-abstractions for continuations are limited. In the following we illustrate these with the specification in Appendix C.

In the denotational definition, the λ-abstractions of *Econt*, *Lcont*, and *Bcont* involve the variables of *Storable-value*, *Location*, and *Tr*. Note that these λ-abstractions are immediately applied the appropriate values if the corresponding program constructs are evaluated normally, and that the bodies of the abstractions are not evaluated before applications. The actual appearances of λ-notations in the valuation functions are limited up to 2. For example, in the valuation function for

E_1+E_2:

$\mathbb{E}[\![E_1 + E_2]\!] = \lambda e.\lambda\alpha.\mathbb{E}[\![E_1]\!]\ e\ (\lambda v_1.\mathbb{E}[\![E_2]\!]\ e\ (\lambda v_2.\alpha\ add(v_1, v_2)))$

the expression continuation for $\mathbb{E}[\![E_1]\!]$ has two occurrences of λ-notations and even if the expression continuation α itself may again contains λ-abstractions with same variables v_1 and v_2, these different occurrences of variables will not interfere because of the scoping. In addition, α will not actually applied to $add(v_1, v_2)$ until $\mathbb{E}[\![E_1]\!]$ and then $\mathbb{E}[\![E_2]\!]$ are successfully evaluated and $add(v_1, v_2)$ is evaluated to a storable-value. These observations motivate us to specify continuations in a partial but complete, with respect to the denotational definition, way.

In the objects TRUTH (*Tr*), LOCATION (*Location*), and STVALUE (*Storable-value*), we divide corresponding sorts into two kinds of subsorts, one for the real elements of the sorts, one for the variables appeared in the denotational definitions. For example, in STVALUE, sorts SValuec and SValuev are declared as subsorts of SValue, where SValuev contains the variables appearing in the denotational definitions: v, v1, v2. In the object CONT that specifies continuations, the applications of Econt, Lcont, and Bcont to the corresponding values are left unspecified. Note also that the sort of command continuation Ccont is specified indirectly, without using of λ-notations. Despite of these, our specification is still complete with respect to the original denotational definition in the sense that reduction of a program (P[_]) always results in a value of sort Answer. This is made possible by introducing auxiliary equations in the object VAL of valuation functions to help the reductions of applications of continuations. For example, in VAL, the equation for valuation function of E_1+E_2 reads:

```
eq : E[ E1 + E2 ] En Ec
     = E[ E1 ] En (& v1 . E[ E2 ] En (& v2 . Ec (v1 + v2))) .
```

where En is an OBJ3 variable of sort Env, and Ec is of sort Ccont. To help the reduction of right-hand expression, we introduce two equations:

```
eq : (& v1 . E[ E ] En (& v2 . Ec (v1 + v2))) V
     = E[ E ] En (& v2 . Ec (V + v2)) .
eq : (& v2 . Ec (V + v2)) V' = Ec (V + V') .
```

where V, V′ are OBJ3 variables of sort SValue (storable values). Actually, the auxiliary equations define completely the functions of continuations. Take expression continuations for example. If we look at the original denotational semantics, there are only five forms of abstractions of the expression continuation:

$\alpha 1(l, c) = \lambda v.\, updatestore(l, v, c)$
where l is a location, c is a command continuation.

$\alpha 2(\alpha) = \lambda v_1.\ \ldots\ (\lambda v_2.\, \alpha\ add(v_1, v_2)) = \lambda v_1.\ \ldots\ \alpha 3(\alpha, v_1)$
$\alpha 3(\alpha, v') = \lambda v.\, \alpha\ add(v', v)$
where α is an expression continuation.

$\alpha 4(\beta) = \lambda v_1.\ \ldots\ (\lambda v_2.\, \beta\ equals(v_1, v_2)) = \lambda v_1.\ \ldots\ \alpha 5(\beta, v_1)$
$\alpha 5(\beta, v') = \lambda v.\, \beta\ equals(v', v)$
where β is a bool-expression continuation.

That is to say, the proper representation of the above five forms of abstractions will constitute the *closure* of the expression continuation. A closure is any data structure that simulates functions. Tuples used in the transformation of defunctionalization are also closures (for a description of closures as representation of higher order functions, one is referred to [R 72]). The two OBJ3 equations presented above actually specified the applications of expression continuations $\alpha 2(\alpha)$ and $\alpha 3(\alpha, v')$ to a value v. In the object VAL, all applications of the above five forms and other forms of continuations are specified by the auxiliary equations.

In order to see how they work, let us reduce an expression for OBJ3. In the following, assume e is of sort Env and

e = ['x (L 0)] (['n 10] ¦),

meaning identifier 'x is a variable denoting location 0, identifier 'n denotes a constant 10; s is of sort Store and s = [(L 0) 3] newstore; and α is some expression continuation.

```
    E[ 'x + 'n + 2 ] e α s
==> E[ 'x ] e (& v1 . E[ 'n + 2 ] e (& v2 . α (v1 + v2))) s
==> (& v1 . E[ 'n + 2 ] e (& v2 . α (v1 + v2))) (access (L 0) s) s
==> (& v1 . E[ 'n + 2 ] e (& v2 . α (v1 + v2))) 3 s
==> E[ 'n + 2 ] e (& v2 . α (3 + v2)) s
==> E[ 'n ] e (& v1 . E[ 2 ] e (& v2 . (& v2 . α (3 + v2)) (v1 + v2))) s
```

```
==> (& v1 . E[ 2 ] e (& v2 . (& v2 . α (3 + v2)) (v1 + v2))) 10 s
==> E[ 2 ] e (& v2 . (& v2 . α (3 + v2)) (10 + v2)) s
==> (& v2 . (& v2 . α (3 + v2)) (10 + v2)) 2 s
==> (& v2 . α (3 + v2)) (10 + 2) s
==> (& v2 . α (3 + v2)) 12 s
==> α (3 + 12) s
==> α 15 s
```

To ensure that OBJ3 will reduce a program to an answer as expected, the assignments of operator precedence and the specifications of evaluation strategies are very critical. Notice that in the objects CONT and VAL all continuation arguments have the strategy of not being evaluated (strat (0)) before being passed to an operation. The continuation arguments act just as place-holders to pass in yet unevaluated portion of a program.

In this section, we have presented an OBJ3 specification of continuation semantics. Because of the features of continuation domains, we did not give a complete specification of λ-reduction for the continuation domains, instead, we give the auxiliary equations in the object VAL of valuation functions to help the valuation of a program construct complete.

Chapter 6
Summary and Extensions

We have described a method of giving algebraic denotational specifications of programming language semantics. Our goal is to follow closest possible to the structure and notations of standard denotational semantics. Thus we are able to provide a tool to test the correctness of semantic definitions by executing OBJ3 specifications.

The major features of our semantic specifications are as follows. (1) We use the initial algebras of specifications as semantic domains in definitions. Thus domain constructions are regarded as set constructions. We have presented methods of specifying compound domains in OBJ3. (2) In our specifications, curried operations in the original denotational semantics are specified in their decurried forms. (3) Since our specifications are first order, we have explored various ways of specifying higher order objects. The most efficient way is defunctionalization that converts higher order objects into first order ones. The specification of a function domain as lambda notations is also possible under certain order of reduction. Since in some cases such as continuation domains, complete specifications of lambda calculus appear very complicated, we tried partial specifications of lambda calculus and used auxiliary equations to help the reduction.

We have given specifications for three different styles of semantic definitions. An obvious extension to this work is include procedures in the language BLOK1 or BLOK2 and give a specification for either direct or continuation semantics. There is no great difficulty in doing this. We can extend the object of denotable values to contain denotations of procedures such as Store -> PostStore in direct semantics, or Ccont -> Ccont in continuation semantics. The case of dynamic scoping can also be specified using appropriate semantic domains.

One interesting future work would be the OBJ3 specification of an implementation from denotational definition for a language. A compiler may be specified in OBJ3 from its denotational semantics using the techniques such as proposed in [Sc 86a] and [W 82].

Given our algebraic denotational specifications of programming language semantics, we may also address the problem of program verification in a way different form Floyd-Hoare's pre- and post- verification conditions. There is some work on the use of OBJ3 in the induction proofs. We hope in the future the problem of program verification based on language's algebraic denotational specification will be studied.

Finally, we found there are two limitations in our algebraic denotational specification of programming languages. First, in our first-order algebraic specifications, although we can conveniently express fixed-point properties of recursively defined functions and domains by equations, we cannot express the minimality property of least-fixed points in first-order logic. This is a central problem of first-order algebraic specifications. The another limitation also relates to the first-orderness. Since our specifications are first-order, some of specifications appears complicated and not neat enough. We feel that a higher order capability in algebraic specification language would make our algebraic denotational specifications more close to the denotational style and more useful as a tool for testing the correctness of semantic definitions.

Appendix A
Direct Semantics Specification for BLOK1

I. Denotational Semantics

Abstract Syntax:

P ∈ Program
D ∈ Declaration
K ∈ Block
C ∈ Command
E ∈ Expression
B ∈ Bool-Expr
I ∈ Identifier
N ∈ Numeral

P ::= begin K end
D ::= D_1 ; D_2 | Var I | Const I N
K ::= let D in C
C ::= C_1 ; C_2 | I := E | while B do C | if B then C_1 else C_2 | K
E ::= E_1 + E_2 | I | N
B ::= E_1 eq E_2 | not B

Semantic Algebras:

(1) Truth Values
Domain t ∈ *Tr* = B
Operations
true, *false* : *Tr*
not : *Tr* → *Tr*

(2) Natural Numbers
Domain n ∈ *Nat* = N
Operations
zero, *one*, *two*,... : *Nat*
plus : *Nat Nat* → *Nat*
equals : *Nat Nat* → *Tr*

(3) Identifiers
Domain i ∈ *Id* = Identifier

```
(4) Storage locations
    Domain l ∈ Location
    Operations
        first-locn : Location
        next-locn : Location → Location

(5) Storable values
    Domain v ∈ Storable-Value = Nat + Uninitialized
                where Uninitialized = Unit
    Operations
        equals : Storable-Value Storable-Value → Tr

        add : Storable-Value Storable-Value → Storable-Value
         add = λv1.λv2. cases(v1) of
                  isNat(n1) → (cases(v2) of
                            isNat(n2) → inNat(n1 plus n2))
                           []isUninitialized( ) → inUnintialized( ) end)
                 []isUninitialized( ) → inUninitialized( ) end

(6) Store
    Domain s ∈ Store = Location → Storable-Value
    Operations
        newstore : Store
         newstore = λl.inUninitialized( )

        access : Location → Store → Storable-Value
         access = λl.λs.s(l)

        update : Location → Storable-Value → Store → Store
         update = λl.λv.λs.[l→v]s

(7) Poststore
    Domain p ∈ Poststore = Store + ErrStore
                where ErrStore = Store
    Operations
        return : Store → Poststore
         return = λs.inStore(s)

        signalerr : Store → Poststore
         signalerr : λs.inErrStore(s)

(8) Denotable values
    Domain d ∈ Denotable-Value = Location + Nat + Undefined
                where Undefined = Unit

(9) Environment
    Domain e ∈ Env = (Id → Denotable-Value) × Location
```

Operations

emptyenv : *Env*
emptyenv = ((λ*i*.in*Undefined*()), *first-locn*)

accessenv : *Id* → *Env* → *Denotable-Value*
accessenv = λ*i*.λ(*map*,*l*).*map*(*i*)

updateenv : *Id* → *Denotable-Value* → *Env* → *Env*
updateenv : λ*i*.λ*d*.λ(*map*,*l*).([*i*→*d*]*map*, *l*)

reserve-locn : *Env* → (*Location* × *Env*)
reserve-locn = λ(*map*,*l*).(*l*, (*map*, *next-locn*(*l*)))

(10) Expressible values

Domain v ∈ *Expressible-Value* = *Storable-Value* + *Errvalue*
where *Errvalue* = *Unit*

(11) Expressible boolean values

Domain b ∈ *Bool-Expr-Value* = *Tr* + *Errbvalue*
where *Errbvalue* = *Unit*

Valuation Functions:

P: Program → *Poststore*
P⟦begin K end⟧ = K⟦K⟧ *emptyenv newstore*

K: Block → *Env* → *Store* → *Poststore*
K⟦let D in C⟧ = λ*e*.C⟦C⟧ (D⟦D⟧ *e*)

D: Declaration → *Env* → *Env*
D⟦D₁ ; D₂⟧ = λ*e*.D⟦D₂⟧ (D⟦D₁⟧ e)

D⟦Var I⟧ = λ*e*.let (*l*′,*e*′)=(*reserve-locn e*)
in (*updateenv* ⟦I⟧ in*Location*(*l*′) *e*′)

D⟦Const I N⟧ = *updateenv* ⟦I⟧ in*Nat*(N⟦N⟧)

C: Command → *Env* → *Store* → *Poststore*
C⟦C₁ ; C₂⟧ = λ*e*.λ*s*. cases (C⟦C₁⟧ *e s*) of
is*Store*(*s*′) → C⟦C₂⟧ *e s*′
⫿is*Errstore*(*s*′) → in*ErrStore*(*s*′) end

C⟦I := E⟧ = λ*e*.λ*s*. cases (*accessenv* ⟦I⟧ *e*) of
is*Location*(*l*) → (cases (E⟦E⟧ *e s*) of
is*Storable-Value*(*v*) → (*return* (*update l v s*))
⫿is*Errvalue*() → (*signalerr s*) end)
⫿is*Nat*(*n*) → (*signalerr s*)

```
                ⟦is Undefined → (signalerr s) end

    C⟦while B do C⟧ = λe.λs. cases (B⟦B⟧ e s) of
                      isTr(t) → (t→(C⟦C ; while B do C⟧ e s)⟦(return s)
                     ⟦isErrbvalue( ) → (signalerr s) end

    C⟦if B then C1 else C2⟧ = λe.λs. cases (B⟦B⟧ e s) of
                      isTr(t) → (t→(C⟦C1⟧ e s)⟦(C⟦C2⟧ e s))
                     ⟦isErrbvalue( ) → (signalerr s) end

    C⟦K⟧ = K⟦K⟧

E: Expression → Env → Store → Expressible-Value
    E⟦E1 + E2⟧ = λe.λs. cases (E⟦E1⟧ e s) of
            isStorable-Value(v1) → (cases (E⟦E2⟧ e s) of
                 isStorable-Value(v2) → inStorable-Value(v1 add v2)
                ⟦isErrvalue( ) → inErrvalue( ) end)
           ⟦isErrvalue( ) → inErrvalue( ) end

    E⟦I⟧ = λe.λs. cases (accessenv ⟦I⟧ e) of
                  isLocation(l) → inStorable-Value(access l s)
                 ⟦isNat(n) → inStorable-Value(inNat(n))
                 ⟦is Undefined → inErrvalue( ) end

    E⟦N⟧ = λe.λs.inStorable-Value(inNat(N⟦N⟧))

B: Bool-Expr → Env → Store → Bool-Expr-Value
    B⟦E1 eq E2⟧ = λe.λs. cases (E⟦E1⟧ e s) of
            isStorable-Value(v1) → (cases (E⟦E2⟧ e s) of
                 isStorable-Value(v2) → inTr(v1 equals v2)
                ⟦isErrvalue( ) → inErrbvalue( ) end)
           ⟦isErrvalue( ) → inErrbvalue( ) end

    B⟦not B⟧ = λe.λs. cases (B⟦B⟧ e s) of
                 isTr(t) → not t
                ⟦isErrbvalue( ) → inErrbvalue( ) end

N: Numeral → Nat   (omitted)
```

II. OBJ3 Specification

```
------------------------ FILE: fun.obj -----------------------------

--- Parameterized object FUN will be used to specify STORE and ENV.
--- Two theories specify the requirements of interfaces

th DOMAIN is
  sort DElt .
  pr BOOL .
  op  _eq_ : DElt DElt -> Bool .
endth

th RANGE is
  sort RElt .
  pr BOOL .
  op ? : -> RElt .
endth

obj FUN [A :: DOMAIN, B :: RANGE] is
  sort Fun .
  op nullF : -> Fun .
  op [__]_ : DElt RElt Fun -> Fun [strat (3 2 1 0)] .
  op __ : Fun DElt -> RElt .
  var x x' : DElt .
  var y : RElt .
  var f : Fun .
  eq : nullF x' = ? .
  eq : ([ x y ] f) x' = if x eq x' then y else f x' fi .
endo

eof


------------------------ FILE: store.obj ---------------------------

in fun

obj UNIT is sort Unit .
  op {} : -> Unit .
endo

--- Specifications of Semantic algebras:
--- Semantic algebras (1)-(3): Truth Values, Natural Numbers,
--- and Identifiers can be specified by built-ins: BOOL, NAT, and QID.

--- (4) Storage locations
obj LOCATION is sort Loc .
```

```
  op first-locn : -> Loc .
  op next-locn_ : Loc -> Loc .
  pr NAT .
  op L_ : Nat -> Loc .
  var N : Nat .
  eq : first-locn = L 0 .
  eq : next-locn (L N) = L (s N) .
endo

--- (5) Storable values
obj STORABLE-VALUE is
  sort Storable-Value .
  pr NAT .
  pr UNIT * (sort Unit to Uninitialized, op ({}) to (uninitialized)) .
  subsorts Nat < Storable-Value .
  subsorts Uninitialized < Storable-Value .
  op _+_ : Storable-Value Storable-Value -> Storable-Value
                                                    [assoc comm] .
  var N : Storable-Value .
  eq : N + uninitialized = uninitialized .
endo

--- Two views of FUN to be used by STORE
view VLOC of LOCATION as DOMAIN is
  sort DElt to Loc .
  var L L' : DElt .
  op Bool : L eq L' to Bool : L == L' .
endv

view VSTVALUE of STORABLE-VALUE as RANGE is
  sort RElt to Storable-Value .
  var N N' : RElt .
  op RElt : ? to Storable-Value : uninitialized .
endv

--- (6) Store
obj STORE is
  pr FUN [VLOC, VSTVALUE]
       * (sort Fun to Store, op (nullF) to (newstore)) .
  op access__ : Loc Store -> Storable-Value .
  op update___ : Loc Storable-Value Store -> Store [strat (3 2 1 0)] .

  var L : Loc .
  var V : Storable-Value .
  var S : Store .
  eq : access L S = S L .
  eq : update L V S = [ L V ] S .
endo
```

```
--- (7) Poststore
obj POSTSTORE is sort PostStore ErrStore .
  pr STORE .
  subsorts Store < PostStore .
  subsorts ErrStore < PostStore .
  op signalerr : Store -> ErrStore .
  op isStore : PostStore -> Bool .
  op isErrStore : PostStore -> Bool .
  var S : Store .
  eq : isStore(S) = true .
  eq : isStore(signalerr(S)) = false .
  eq : isErrStore(S) = false .
  eq : isErrStore(signalerr(S)) = true .
endo

--- (8) Denotable values
obj DENOTABLE-VAL is sort Denot-Val .
  pr NAT .
  pr LOCATION .
  pr UNIT * (sort Unit to Undefined, op ({}) to (undefined)) .
  subsorts Loc < Denot-Val .
  subsorts Nat < Denot-Val .
  subsorts Undefined < Denot-Val .
  op isLoc : Denot-Val -> Bool .
  op isNat : Denot-Val -> Bool .
  op isUndefined : Denot-Val -> Bool .
  op selecNat : Denot-Val -> Nat .

  var D : Denot-Val .
  var L : Loc .
  var N : Nat .
  var U : Undefined .
  eq : isLoc(L) = true .
  eq : isLoc(N) = false .
  eq : isLoc(U) = false .
  eq : isNat(N) = true .
  eq : isNat(L) = false .
  eq : isNat(U) = false .
  eq : isUndefined(D) = D == undefined .
  eq : selecNat(N) = N .
endo

--- Two views used in ENV
view VID of QID as DOMAIN is
  var I I' : DElt .
  op Bool : I eq I' to Bool : I == I' .
endv
```

```
view VDEN of DENOTABLE-VAL as RANGE is
  var V V' : RElt .
  op RElt : ? to Denot-Val : undefined .
endv

--- (9) Environment
obj ENV is sort Env .
  pr FUN [VID, VDEN] * (sort Fun to Idmap, op (nullF) to (¦)) .
  op <_,_> : Idmap Loc -> Env .
  op emptyenv : -> Env .
  op updateenv___ : Id Denot-Val Env -> Env [strat (3 2 1 0)] .
  op accessenv__ : Id Env -> Denot-Val [strat (2 1 0) memo] .
  op reserve-locn_ : Env -> Env .
  op get-locn_ : Env -> Loc .

  var L : Loc .
  var I : Id .
  var D : Denot-Val .
  var M : Idmap .
  eq : emptyenv = < ¦ , first-locn > .
  eq : updateenv I D < M , L > = < [ I D ] M , L > .
  eq : accessenv I < M , L > = M I .
  eq : reserve-locn < M , L > = < M , next-locn L > .
  eq : get-locn < M , L > = L .
endo

--- (10) Expressible values: Expressible-Value
obj EVALUE is sort Evalue .
  pr STORABLE-VALUE .
  pr UNIT * (sort Unit to Errvalue, op ({}) to (errvalue)) .
  subsorts Storable-Value < Evalue .
  subsorts Errvalue < Evalue .
  op isStorable-Value : Evalue -> Bool .
  op isErrvalue : Evalue -> Bool .
  var V : Storable-Value .
  var E : Evalue .
  eq : isStorable-Value(V) = true .
  eq : isStorable-Value(errvalue) = false .
  eq : isErrvalue(E) = E == errvalue .
endo

--- (11) Expressible boolean values: Bool-Expr-Value
obj BVALUE is sort Bvalue .
  pr BOOL .
  pr UNIT * (sort Unit to Errvalue, op ({}) to (errbvalue)) .
  subsorts Bool < Bvalue .
  subsorts Errvalue < Bvalue .
```

```
  op isTr : Bvalue -> Bool .
  op isErrvalue : Bvalue -> Bool .
  var T : Bool .
  var B : Bvalue .
  eq : isTr(T) = true .
  eq : isTr(errbvalue) = false .
  eq : isErrvalue(B) = B == errbvalue .
endo


--- Syntactic domain from abstract syntax
obj SYN-DOM is
  sorts Prog Decl Com Block Expr Bexpr .
  protecting NAT .
  protecting QID .
  subsorts Nat < Expr .
  subsorts Id < Expr .
  subsorts Block < Com .

  op begin_end : Block -> Prog [prec 9] .

  op _;_ : Decl Decl -> Decl [assoc prec 6] .
  op Var_ : Id -> Decl [prec 5] .
  op Const__ : Id Nat -> Decl [prec 5] .

  op let_in_ : Decl Com -> Block [prec 8] .

  op _;_ : Com Com -> Com [assoc prec 7] .
  op _:=_ : Id Expr -> Com [prec 5] .
  op while_do_ : Bexpr Com -> Com [prec 5] .
  op if_then_else_ : Bexpr Com Com -> Com [prec 5] .

  op _+_ : Expr Expr -> Expr [prec 3] .

  op _eq_ : Expr Expr -> Bexpr [prec 4] .
  op not_ : Bexpr -> Bexpr [prec 4] .
endo


--- Valuation functions
obj VAL-FUN is
  pr POSTSTORE .
  pr ENV .
  pr EVALUE .
  pr BVALUE .
  pr SYN-DOM .
  op P[_] : Prog -> PostStore .
  op K[_]__ : Block Env Store -> PostStore .
```

```
op D[_]_ : Decl Env -> Env .
op C[_]__ : Com Env Store -> PostStore [memo] .
op E[_]__ : Expr Env Store -> Evalue [memo] .
op B[_]__ : Bexpr Env Store -> Bvalue [memo] .

var I : Id .
var N : Nat .
var D D1 D2 : Decl .
var K : Block .
var E E1 E2 : Expr .
var B : Bexpr .
var C C1 C2 : Com .
var e : Env .
var s : Store .

eq : P[ begin K end ] = K[ K ] emptyenv newstore .

eq : K[ let D in C ] e s = C[ C ] (D[ D ] e) s .

eq : D[ Var I ] e = updateenv I (get-locn e) (reserve-locn e) .
eq : D[ Const I N ] e = updateenv I N e .
eq : D[ D1 ; D2 ] e = D[ D2 ] (D[ D1 ] e) .

eq : C[ C1 ; C2 ] e s = if isStore(C[ C1 ] e s)
                        then C[ C2 ] e (C[ C1 ] e s)
                        else signalerr(s) fi .
eq : C[ I := E ] e s
      = if isLoc(accessenv I e) and isStorable-Value(E[ E ] e s)
        then update (accessenv I e) (E[ E ] e s) s
        else signalerr(s) fi .

eq : C[ while B do C ] e s
      = if isTr(B[ B ] e s)
        then if (B[ B ] e s) then C[ C ; while B do C ] e s
                             else s fi
        else signalerr(s) fi .

eq : C[ if B then C1 else C2 ] e s
      = if isTr(B[ B ] e s)
        then if (B[ B ] e s) then C[ C1 ] e s
                             else C[ C2 ] e s fi
        else signalerr(s) fi .

eq : C[ K ] e s = K[ K ] e s .

ceq : E[ I ] e s = access (accessenv I e) s if isLoc(accessenv I e) .
ceq : E[ I ] e s = selecNat(accessenv I e) if isNat(accessenv I e) .
ceq : E[ I ] e s = errvalue if isUndefined(accessenv I e) .
```

```
  eq : E[ N ] e s = N .
  eq : E[ E1 + E2 ] e s
       = if isStorable-Value(E[ E1 ] e s)
            and isStorable-Value(E[ E2 ] e s)
         then (E[ E1 ] e s) + (E[ E2 ] e s)
         else errvalue fi .

  eq : B[ E1 eq E2 ] e s
       = if isStorable-Value(E[ E1 ] e s)
            and isStorable-Value(E[ E2 ] e s)
         then (E[ E1 ] e s) == (E[ E2 ] e s)
         else errbvalue fi .
  eq : B[ not B ] e s = if isTr(B[ B ] e s) then not (B[ B ] e s)
                        else errbvalue fi .
endo

eof
```

III. Example Runs

```
Welcome to OBJ3 Version .99
system built: (1988 3 15 15 5 55)
Copyright 1987 by the OBJ3 Group (KF, JAG, JPJ, JM, TW, CK, HK, AM)
OBJ3> in blok1
=========================================
in fun
Reading in file : "fun"
=========================================
th DOMAIN
=========================================
th RANGE
=========================================
obj FUN
Done reading in file: "fun"
=========================================
obj UNIT
=========================================
obj LOCATION
=========================================
obj STORABLE-VALUE
=========================================
view VLOC
=========================================
view VSTVALUE
=========================================
obj STORE
```

```
=========================================
obj POSTSTORE
=========================================
obj DENOTABLE-VAL
=========================================
view VID
=========================================
view VDEN
=========================================
obj ENV
=========================================
obj EVALUE
=========================================
obj BVALUE
=========================================
obj SYN-DOM
=========================================
obj VAL-FUN
OBJ3> reduce in VAL-FUN as :
  P[ begin
        let
            Var 'x ; Var 'y ; Const 'one 1
        in
            'x := 'one ;
            'y := 'x + 'one
     end ] .
reducing term:  (P[ (begin (let ((Var 'x) ; ((Var 'y) ; (Const 'one 1)))
in (('x := 'one) ; ('y := ('x + 'one)))) end) ])
reduction result Store: ([ (L 1) 2 ] ([ (L 0) 1 ] newstore))

OBJ3> reduce in VAL-FUN as :
  P[ begin
        let
            Var 'sum ; Var 'i
        in
            'sum := 0 ;
            'i := 0 ;
            while not ('i eq 3)
            do ('i := 'i + 1 ; 'sum := 'sum + 'i)
     end ] .
reducing term: (P[ (begin (let ((Var 'sum) ; (Var 'i)) in (('sum := 0) ;
(('i :=  0) ;  (while (not  ('i eq  3)) do  (('i := ('i + 1)) ; ('sum :=
('sum + 'i))))))) end) ])
reduction result Store: ([ (L 0) 6 ] ([ (L 1) 3 ] ([ (L 0) 3 ]  ([ (L 1)
2 ] ([ (L 0) 1 ] ([ (L 1) 1 ] ([ (L 1) 0 ] ([ (L 0) 0 ] newstore)))))))
```

```
OBJ3> reduce in VAL-FUN as :
  P[ begin
        let
            Var 'x ; Const 'n 1 ; Var 'y
        in
            'x := 'y + 1 ;
            'n := 'x + 1 ;
            if 'x eq 1 then 'x := 10 else 'x := 0
     end ] .
reducing term: (P[ (begin (let ((Var 'x) ; ((Const 'n 1) ; (Var 'y))) in
(('x := ('y + 1)) ; (('n := ('x + 1)) ; (if ('x  eq 1)  then ('x  := 10)
else ('x := 0))))) end) ])
reduction result ErrStore: signalerr(([ (L 0) uninitialized ] newstore))

OBJ3> reduce in VAL-FUN as :
        P[ begin
              let Var 'x ; Var 'y
              in 'x := 1 ;
                 (let Const 'y 10 in 'x := 'y + 'x) ;
                 'y := 'x
           end ] .
reducing term:  (P[ (begin  (let ((Var  'x) ;  (Var 'y)) in (('x := 1) ;
((let (Const 'y 10) in ('x := ('y + 'x))) ; ('y := 'x)))) end) ])
reduction result Store: ([ (L 1) 11 ] ([  (L  0)  11  ]  ([  (L  0)  1 ]
newstore)))
OBJ3>
```

Appendix B
Semantic Specification For PLISP

I. Denotational Semantics

Abstract Syntax:

E ∈ Expression
N ∈ Numeral
I ∈ Identifier

E ::= LET I BE E_1 IN E_2 ¦ LAMBDA I E ¦ E_1 E_2 ¦
E_1 CONS E_2 ¦ HEAD E ¦ TAIL E ¦ NIL ¦ I ¦ N

Semantic Algebras:

(1) Natural Numbers
Domain n ∈ *Nat* = **N**

(2) Identifiers
Domain i ∈ *Id* = Identifier

(3) Denotable values, functions, and lists
Domains d ∈ *Denotable-value* = *Function* + *List* + *Nat* + *Error*
where *Error* = *Unit*
f ∈ *Function* = *Denotable-value* → *Denotable-value*
l ∈ *List* = *Nil* + *NeList*
where *Nil* = *Unit*, and *NeList* = *Denotable-value* × *List*
Operations
hd : *List* → *Denotable-value*
hd = λ*l*. cases (*l*) of is*Nil*() → in*Error*()
[]is*NeList*((*d*,*l'*)) → *d* end

tl : *List* → *List*
tl = λ*l*. cases (*l*) of is*Nil*() → in*Error*()
[]is*NeList*((*d*,*l'*)) → *l'* end

cons : *Denotable-value List* → *List*
cons = λ*d*.λ*l*.in*List*((*d*,*l*))

(4) Environments
Domain e ∈ *Env* = *Id* → *Denotable-value*

Operations

emptyenv : *Env*
emptyenv = $\lambda i.$in*Error*( )

accessenv : *Id* $\rightarrow$ *Env* $\rightarrow$ *Denotable-value*
accessenv = $\lambda i.\lambda e.e(i)$

updateenv : *Id* $\rightarrow$ *Denotable-value* $\rightarrow$ *Env* $\rightarrow$ *Env*
updateenv = $\lambda i.\lambda d.\lambda e.\lceil i \mapsto d \rceil e$

(5) Expressible values
Domain v $\in$ *Expressible-value* = *Denotable-value*

Valuation Functions:
E : Expression $\rightarrow$ *Env* $\rightarrow$ *Expressible-value*
E⟦LET I BE E_1 IN E_2⟧ = $\lambda e.$E⟦E_2⟧(*updateenv* ⟦I⟧ (E⟦E_1⟧e) *e*)

E⟦LAMBDA I E⟧ = $\lambda e.$in*Function*($\lambda d.$E⟦E⟧(*updateenv* ⟦I⟧ *d e*))

E⟦E_1 E_2⟧ = $\lambda e.$ let *v* = (E⟦E_1⟧e) in cases *v* of
is*Function*(*f*) $\rightarrow$ *f*(E⟦E_2⟧e)
⟦is*List*(*l*) $\rightarrow$ in*Error*( )
⟦is*Nat*(*n*) $\rightarrow$ in*Error*( ) ⟦is*Error*( ) $\rightarrow$ in*Error*( ) end

E⟦E_1 CONS E_2⟧ = $\lambda e.$ let *v* = (E⟦E_2⟧e) in cases *v* of
is*Function*(f) $\rightarrow$ in*Error*( )
⟦is*List*(*l*) $\rightarrow$ in*List*((E⟦E_1⟧e) *cons l*)
⟦is*Nat*(*n*) $\rightarrow$ in*Error*( ) ⟦is*Error*( ) $\rightarrow$ in*Error*( ) end

E⟦HEAD E⟧ = $\lambda e.$ let *v* = (E⟦E⟧e) in cases *v* of
is*Function*(*f*) $\rightarrow$ in*Error*( )
⟦is*List*(*l*) $\rightarrow$ *hd l*
⟦is*Nat*(*n*) $\rightarrow$ in*Error*( ) ⟦is*Error*( ) $\rightarrow$ in*Error*( ) end

E⟦TAIL E⟧ = $\lambda e.$ let *v* = (E⟦E⟧e) in cases *v* of
is*Function*(*f*) $\rightarrow$ in*Error*( )
⟦is*List*(*l*) $\rightarrow$ in*List*(*tl l*)
⟦is*Nat*(*n*) $\rightarrow$ in*Error*( ) ⟦is*Error*( ) $\rightarrow$ in*Error*( ) end

E⟦NIL⟧ = $\lambda e.$in*List*(*Nil*)

E⟦I⟧ = $\lambda e.$*accessenv* ⟦I⟧ *e*

E⟦N⟧ = $\lambda e.$N⟦N⟧

N: Numeral $\rightarrow$ *Nat* (omitted)

II. OBJ3 Specification

```
----------------------  FILE: plisp.obj  ---------------------------

--- Variables used in the λ-abstractions of denotable values.

obj VARS is sort Vars .
  pr NAT .
  op v_ : Nat -> Vars .
  op nextVar : Vars -> Vars .
  op same : Vars Vars -> Bool .
  var N N' : Nat .
  eq : nextVar (v N) = v (s N) .
  eq : same((v N), (v N')) = N == N' .
endo

--- Domain of denotable values.

obj DENOT-VAL is sort Den .
  pr NAT .
  pr VARS .
  subsorts Nat < Den .
  subsorts Vars  < Den .
  op Err : -> Den .
  op Nil : -> Den .

  op _;_ : Den Den -> Den [assoc] .
  op hd_ : Den -> Den .
  op tl_ : Den -> Den .

  op &_._ : Vars Den -> Den .
  op __ : Den Den -> Den .

  op [_/_]_ : Den Vars Den -> Den [memo] .
  op red : Den -> Den .

  op isList : Den -> Bool .
  op isFunc : Den -> Bool .
  op isErr : Den -> Bool .

  op isVarterm : Den -> Bool .

  var X Y : Vars .
  var D D' L F : Den .
  var N : Nat .

  eq : hd (D ; L) = D .
  eq : hd Nil = Err .
```

```
  eq : tl (D ; L) = L .
  eq : tl Nil = Err .

  eq : [ D / X ] N = N .
  ceq : [ D / X ] Y = D if same(X, Y) .
  ceq : [ D / X ] Y = Y if not same(X, Y) .
---  eq : [ D / X ] Y = if same(X, Y) then D else Y fi . DID'T WORK
  eq : [ D / X ] (F D')
       = if isFunc([ D / X ] F)
         then ([ D / X ] F) ([ D / X ] D') else Err fi .
  eq : [ D / X ] (& Y . D') = if same(X, Y) then (& Y . D')
                              else (& Y . ([ D / X ] D')) fi .
  eq : [ D / X ] (D' ; L)
       = if isList([ D / X ] L)
         then ([ D / X ] D') ; ([ D / X ] L) else Err fi .
  eq : [ D / X ] Nil = Nil .
  eq : [ D / X ] (hd L) = if isList([ D / X ] L)
                          then hd ([ D / X ] L) else Err fi .
  eq : [ D / X ] (tl L) = if isList([ D / X ] L)
                          then tl ([ D / X ] L) else Err fi .
  eq : [ D / X ] Err = Err .

  eq : red((& X . D) D') = red([ D' / X ] D) .
  eq : red(X D) = X D .
  eq : red(N) = N .
  eq : red(Nil) = Nil .
  eq : red(D ; L) = D ; L .
  eq : red(tl L) = tl L .
  eq : red(hd L) = hd L .
  eq : red(& X . D) = (& X . red(D)) .
  eq : red(Err) = Err .

  eq : isList(D ; L) = isList(L) .
  eq : isList(Nil) = true .
  eq : isList(N) = false .
  eq : isList(Err) = false .
  eq : isList(& X . D) = false .
  ceq : isList(D) = true if isVarterm(D) .

  eq : isFunc(& X . D) = true .
  eq : isFunc(Nil) = false .
  eq : isFunc(D ; L) = false .
  eq : isFunc(N) = false .
  eq : isFunc(Err) = false .
  ceq : isFunc(D) = true if isVarterm(D) .

  eq : isErr(D) = D == Err .
```

```
  eq : isVarterm(X) = true .
  eq : isVarterm(X D) = true .
  eq : isVarterm(hd D) = isVarterm(D) .
  eq : isVarterm(tl D) = isVarterm(D) .
endo

in fun      --- Specification of parameterized function domains

view VID of QID as DOMAIN is
  var I I' : DElt .
  op Bool : I eq I' to Bool : I == I' .
endv

view VD of DENOT-VAL as RANGE is
  sort RElt to Den .
  op RElt : ? to Den : Err .
endv

obj ENV is sort Env .
  pr FUN [VID, VD] * (sort Fun to Map, op (nullF) to (¦)) .
  op <_,_> : Map Vars -> Env .
  op emptyenv : -> Env .
  op accessenv__ : Id Env -> Den .
  op updateenv___ : Id Den Env -> Env .
  op getvar_ : Env -> Vars .
  op gonextvar_ : Env -> Env .
  var I : Id .
  var X : Vars .
  var M : Map .
  var D : Den .
  eq : emptyenv = < ¦ , (v 0) > .
  eq : accessenv I < M , X > = M I .
  eq : updateenv I D < M , X > = < ([ I D ] M) , X > .
  eq : getvar < M , X > = X .
  eq : gonextvar < M , X > = < M , nextVar(X) > .
endo

obj EXPRESSION is sort Expr .
  pr NAT .
  pr QID .
  subsorts Id < Expr .
  subsorts Nat < Expr .
  op NIL : -> Expr .
  op HEAD_ : Expr -> Expr [prec 3] .
  op TAIL_ : Expr -> Expr [prec 3] .
  op _CONS_ : Expr Expr -> Expr [assoc prec 4] .
  op __ : Expr Expr -> Expr [prec 6] .
  op LAMBDA__ : Id Expr -> Expr [prec 5] .
```

```
  op LET_BE_IN_ : Id Expr Expr -> Expr [prec 7] .
endo

obj VAL is
  pr ENV .
  pr EXPRESSION .
  op E[_]_ : Expr Env -> Den [strat(2 0) memo] .

  var I : Id .
  var E E1 E2 : Expr .
  var e : Env .
  var N : Nat .

  eq : E[ I ] e = accessenv I e .
  eq : E[ N ] e = N .
  eq : E[ NIL ] e = Nil .
  eq : E[ HEAD E ] e
       = if isList(E[ E ] e) then hd (E[ E ] e) else Err fi .
  eq : E[ TAIL E ] e
       = if isList(E[ E ] e) then tl (E[ E ] e) else Err fi .
  eq : E[ E1 CONS E2 ] e
       = if isList(E[ E2 ] e) and (not isErr(E[ E1 ] e))
         then (E[ E1 ] e) ; (E[ E2 ] e) else Err fi .
  eq : E[ E1 E2 ] e
       = if isFunc(E[ E1 ] e) and (not isErr(E[ E2 ] e))
         then red((E[ E1 ] e) (E[ E2 ] e)) else Err fi .
  eq : E[ LAMBDA I E ] e
       = (& (getvar e) .
              (E[ E ] (updateenv I (getvar e) (gonextvar e)))) .
  eq : E[ LET I BE E1 IN E2 ] e
       = if isErr(E[ E1 ] e) then Err
         else E[ E2 ] (updateenv I (E[ E1 ] e) e) fi .
endo

eof
```

III. Example Runs

```
Welcome to OBJ3 Version .99
system built: (1988 3 15 15 5 55)
Copyright 1987 by the OBJ3 Group (KF, JAG, JPJ, JM, TW, CK, HK, AM)
OBJ3> in plisp
=========================================
obj VARS
=========================================
obj DENOT-VAL
=========================================
in fun
Reading in file : "fun"
=========================================
th DOMAIN
=========================================
th RANGE
=========================================
obj FUN
Done reading in file: "fun"
=========================================
view VID
=========================================
view VD
=========================================
obj ENV
=========================================
obj EXPRESSION
=========================================
obj VAL

OBJ3> reduce in VAL as :
  E[ LAMBDA 'f (LAMBDA 'x ('f (HEAD 'x))) ] emptyenv
.
reducing term: (E[ (LAMBDA 'f (LAMBDA 'x ('f (HEAD 'x)))) ] emptyenv)
reduction result Den: (& (v 0) . (& (v 1) . ((v 0) (hd (v 1)))))

OBJ3> reduce in VAL as :
  E[ (LAMBDA 'f (LAMBDA 'x ('f (HEAD 'x))))
            (LAMBDA 'x ('x CONS NIL)) ] emptyenv
.
reducing term:  (E[ ((LAMBDA  'f (LAMBDA  'x ('f (HEAD 'x)))) (LAMBDA 'x
('x CONS NIL))) ] emptyenv)
reduction result Den: (& (v 1) . ((hd (v 1)) ; Nil))
```

```
OBJ3> reduce in VAL as :
  E[ LET 'snd BE LAMBDA 'x (HEAD (TAIL 'x)) IN
                'snd (2 CONS 3 CONS NIL) ] emptyenv
.
reducing term: (E[ (LET 'snd BE (LAMBDA 'x (HEAD (TAIL 'x))) IN ('snd (2
 CONS (3 CONS NIL)))) ] emptyenv)
reduction result Nat: 3

OBJ3> reduce in VAL as :
  E[ LET 'f BE 0 IN
       LET 'f BE LAMBDA 'z ('f CONS 'z) IN
          LET 'z BE 1 IN 'f ('z CONS NIL) ] emptyenv
.
reducing term: (E[ (LET 'f BE 0 IN (LET 'f BE (LAMBDA 'z ('f CONS 'z))
 IN (LET 'z BE 1 IN ('f ('z CONS NIL))))) ] emptyenv)
reduction result Den: (0 ; (1 ; Nil))

OBJ3> reduce in VAL as :
  E[ LET 'g BE LAMBDA 'f (LAMBDA 'x ('f 'x)) IN
       LET 'list BE LAMBDA 'x ('x CONS NIL) IN
           ('g 'list) 2 ] emptyenv
.
reducing term: (E[ (LET 'g BE (LAMBDA 'f (LAMBDA 'x ('f 'x))) IN (LET
 'list BE (LAMBDA 'x ('x CONS NIL)) IN (('g 'list) 2))) ] emptyenv)
reduction result Den: (2 ; Nil)

OBJ3> reduce in VAL as :
  E[ LET 'f BE LAMBDA 'x (HEAD 'x) IN 'f 2 ] emptyenv
.
reducing term: (E[ (LET 'f BE (LAMBDA 'x (HEAD 'x)) IN ('f 2)) ]
 emptyenv)
reduction result Den: Err

OBJ3> reduce in VAL as :
  E[ (LAMBDA 'f ('f 2)) NIL ] emptyenv
.
reducing term: (E[ ((LAMBDA 'f ('f 2)) NIL) ] emptyenv)
reduction result Den: Err


OBJ3> reduce in VAL as :
  E[ LET 'f BE LAMBDA 'x (HEAD 'x) IN 'f 2 ] emptyenv
.
reducing term: (E[ (LET 'f BE (LAMBDA 'x (HEAD 'x)) IN ('f 2)) ]
 emptyenv)
reduction result Den: Err
```

Appendix C
Continuation Semantics Specification for BLOK2

I. Denotational Semantics

Abstract Syntax:

P ∈ Program
K ∈ Block
D ∈ Declaration
C ∈ Command
E ∈ Expression
B ∈ Bool-Expr
I ∈ Identifier
N ∈ Numeral

P ::= begin K end
K ::= let D in C
D ::= D_1 ; D_2 ¦ Var I ¦ Const I N
C ::= C_1 ; C_2 ¦ I := E ¦ while B do C ¦ if B then C_1 else C_2 ¦ stop
E ::= E_1 + E_2 ¦ I ¦ N
B ::= E_1 eq E_2 ¦ not B

Semantics Algebras:

(1)-(3) Truth values, Natural numbers, Identifiers
(same as defined in Appendix A)

(4)-(6) Storage locations, Storable values, Stores
(same as defined in Appendix A)

(7)-(8) Denotable values, Environments
(same as defined in Appendix A)

(9) Messages
Domain m ∈ *Message*
Operations
normal : *Message*

stopped : *Message*
id-use-err : *Message*
id-undefined : *Message*

(10) Answers
Domain a ∈ *Answer* = *Message* × *Store*

(11) Command continuations
Domain c ∈ *Ccont* = *Store* → *Answer*
Operations
terminate : *Message* → *Ccont*
terminate = λ*m*.λ*s*.(*m*,*s*)

updatestore : *Location* → *Storable-Value* → *Ccont* → *Ccont*
updatestore = λ*l*.λ*v*.λ*c*.λ*s*.*c*(*update l v s*)

(12) Expression continuations
Domain α ∈ *Econt* = *Storable-Value* → *Ccont*

(13) Identifier continuations
Domain β ∈ *Lcont* = *Location* → *Ccont*

(14) Bool-expression continuations
Domain τ ∈ *Bcont* = *Tr* → *Ccont*

Valuation Functions:

P: Program → *Answer*
P⟦begin K end⟧ = K⟦K⟧ *emptyenv* (*terminate normal*) *newstore*

K: Block → *Env* → *Ccont* → *Ccont*
K⟦let D in C⟧ = λ*e*.λ*c*.C⟦C⟧ (D⟦D⟧ *e*) *c*

D: Declaration → *Env* → *Env*
D⟦D₁ ; D₂⟧ = λ*e*.D⟦D₂⟧(D⟦D₁⟧*e*)

D⟦Var I⟧ = λ*e*.let (*l*′, *e*′) = *reserve-locn e*′
in (*updateenv* ⟦I⟧ in*Location*(*l*′) *e*′)

D⟦Const I N⟧ = *updateenv* ⟦I⟧ in*Nat*(N⟦N⟧)

C: Command → *Env* → *Ccont* → *Ccont*
C⟦C₁ ; C₂⟧ = λ*e*.λ*c*.C⟦C₁⟧ *e* (C⟦C₂⟧ *e c*)

C⟦I := E⟧ = λ*e*.λ*c*.L⟦I⟧ *e* (λ*l*.E⟦E⟧ *e* (λ*v*. *updatestore l v c*))

C⟦while B do C⟧ = *wh*

$wh = \lambda e.\lambda c.B[\![B]\!]\ e\ (\lambda t.\ t \rightarrow C[\![C]\!]\ e\ (wh\ e\ c)\ []\ c)$

$C[\![$if B then C_1 else $C_2]\!]$
$= \lambda e.\lambda c.B[\![B]\!]\ e\ (\lambda t.t \rightarrow C[\![C_1]\!]\ e\ c\ []\ C[\![C_2]\!]\ e\ c)$

$C[\![$stop$]\!] = \lambda e.\lambda c.$ *terminate stopped*

E: Expression $\rightarrow$ *Env* $\rightarrow$ *Econt* $\rightarrow$ *Ccont*
$E[\![E_1$ + $E_2]\!] = \lambda e.\lambda\alpha.E[\![E_1]\!]\ e\ (\lambda v_1.E[\![E_2]\!]\ e\ (\lambda v_2.\alpha\ add(v_1, v_2)))$

$E[\![I]\!] = \lambda e.\lambda\alpha.$ cases $(accessenv\ [\![I]\!]\ e)$ of
isLocation$(l) \rightarrow \lambda s.\alpha(access\ l\ s)$
$[]$*isNat*$(n) \rightarrow \alpha\ n$
$[]$*isUndefined*$() \rightarrow$ *terminate id-undefined* end

$E[\![N]\!] = \lambda e.\lambda\alpha.\ \alpha\ N[\![N]\!]$

B: Bool-Expr $\rightarrow$ *Env* $\rightarrow$ *Bcont* $\rightarrow$ *Ccont*
$B[\![E_1$ eq $E_2]\!] = \lambda e.\lambda\tau.E[\![E_1]\!]\ e\ (\lambda v_1.E[\![E_2]\!]\ e\ (\lambda v_2.\tau\ (equals\ v_1\ v_2)))$

$B[\![$not B$]\!] = \lambda e.\lambda\tau.B[\![B]\!]$ e $(\lambda t.\tau(not\ t))$

L: Identifier $\rightarrow$ *Env* $\rightarrow$ *Lcont* $\rightarrow$ *Ccont*
$L[\![I]\!] = \lambda e.\lambda\beta.$ cases $(accessenv\ [\![I]\!]\ e)$ of
isLocation$(l) \rightarrow \beta\ l$
$[]$*isNat*$(n) \rightarrow$ *terminate id-use-err*
$[]$*isUndefined* $\rightarrow$ *terminate id-undefined* end

II. OBJ3 Specification

```
------------------------- FILE: blok2.obj -------------------------

in fun            --- Specification of parameterized function domain

--- (1) Truth values
obj TRUTH is sort Tr Trv .
  subsorts Trv < Tr .
  pr BOOL .
  subsorts Bool < Tr .
  op t : -> Trv .
  op not_ : Tr -> Tr .
endo

--- (4) Storage locations
obj LOCATION is sort Loc Locc Locv .
  subsorts Locc < Loc .
```

```
  subsorts Locv < Loc .
  op first-locn : -> Locc .
  op next-locn_ : Locc -> Locc .
  pr NAT .
  op L_ : Nat -> Locc .

  op l : -> Locv .

  var N : Nat .
  eq : first-locn = L 0 .
  eq : next-locn (L N) = L (s N) . ,
endo

--- (5) Storable values
obj STVALUE is sorts SValue SValuec SValuev .
  subsorts SValuec < SValue .
  subsorts SValuev < SValue .
  pr NAT .
  subsorts Nat < SValuec .
  op equals : SValue SValue -> Bool .
  op uninitialized : -> SValuec .

  op v : -> SValuev .
  op v1 : -> SValuev .
  op v2 : -> SValuev .

  op _+_ : SValue SValue -> SValue [assoc comm] .

  var V : SValue .
  eq : V + uninitialized = uninitialized .
  var V1 V2 : SValuec .
  eq : equals(V1, V2) = V1 == V2 .
endo

--- Two views for STORE
view VLOC of LOCATION as DOMAIN is
  sort DElt to Locc .
  var L L' : DElt .
  op Bool : L eq L' to Bool : L == L' .
endv

view VSTVALUE of STVALUE as RANGE is
  sort RElt to SValuec .
  var N N' : RElt .
  op RElt : ? to SValue : uninitialized .
endv
```

```
--- (6) Stores
obj STORE is
  pr FUN [VLOC, VSTVALUE]
                * (sort Fun to Store, op (nullF) to (newstore)) .
  op access__ : Loc Store -> SValuec .
  op update___ : Loc SValue Store -> Store [strat (3 2 1 0)] .

  var L : Locc .
  var V : SValuec .
  var S : Store .
  eq : access L S = S L .
  eq : update L V S = [ L V ] S .
endo

--- (9) Messages
obj MESGS is sort Mesg .
  op normal : -> Mesg .
  op stopped : -> Mesg .
  op id-use-err : -> Mesg .
  op id-undefined : -> Mesg .
endo

--- (10) Answers
obj ANSWER is sort Answer .
  pr STORE .
  pr MESGS .
  op <_,_> : Mesg Store -> Answer .
endo

--- (11)-(14) Continuations
obj CONT is sort Ccont Bcont Econt Lcont .
  pr STORE .
  pr ANSWER .
  pr TRUTH .
  op __ : Ccont Store -> Answer [strat (2 1 0) prec 9] .
  op terminate_ : Mesg -> Ccont [prec 2] .

  op updatestore___ : Ccont Loc SValue -> Ccont [strat (2 3 0)] .

  op &_._ : SValuev Ccont -> Econt [strat (0) prec 8] .
  op __ : Econt SValue -> Ccont [strat (2 0) prec 7] .

  op &_._ : Locv Ccont -> Lcont [strat (0) prec 8] .
  op __ : Lcont Loc -> Ccont [strat (2 0) prec 7] .

  op &_._ : Trv Ccont -> Bcont [strat (0) prec 8] .
  op __ : Bcont Tr -> Ccont [strat (2 0) prec 7] .
```

```
  op _=>_¦_ : Tr Ccont Ccont -> Ccont [strat (0) prec 10] .

  var M : Mesg .
  var S : Store .
  eq : terminate M S = < M , S > .

  var C : Ccont .
  var V : SValuec .
  var L : Locc .
  eq : (updatestore C L V) S = C (update L V S) .

  var B : Bool .
  var C1 C2 : Ccont .
  eq : B => C1 ¦ C2 = if B then C1 else C2 fi .
endo

obj UNIT is sort Unit .
  op {} : -> Unit .
endo

--- (7) Denotable values
obj DEN-VAL is sort Denotval .
  pr LOCATION .
  pr NAT .
  pr UNIT * (sort Unit to Undefined, op ({}) to (undefined)) .
  subsorts Locc < Denotval .
  subsorts Nat < Denotval .
  subsorts Undefined < Denotval .
  op isLoc : Denotval -> Bool .
  op isNat : Denotval -> Bool .
  op isUndefined : Denotval -> Bool .
  op selecNat : Denotval -> Nat .
  op selecLoc : Denotval -> Locc .

  var D : Denotval .
  var L : Locc .
  var N : Nat .
  var U : Undefined .
  eq : isLoc(L) = true .
  eq : isLoc(N) = false .
  eq : isLoc(U) = false .
  eq : isNat(N) = true .
  eq : isNat(L) = false .
  eq : isNat(U) = false .
  eq : isUndefined(D) = D == undefined .
  eq : selecNat(N) = N .
  eq : selecLoc(L) = L .
endo
```

```
view VID of QID as DOMAIN is
  var I I' : DElt .
  op Bool : I eq I' to Bool : I == I' .
endv

view VDEN of DEN-VAL as RANGE is
  var V V' : RElt .
  op RElt : ? to Denotval : undefined .
endv

--- (8) Environment
obj ENV is sort Env .
  pr FUN [VID, VDEN] * (sort Fun to Idmap, op (nullF) to (¦)) .
  op <_,_> : Idmap Locc -> Env .
  op emptyenv : -> Env .
  op updateenv___ : Id Denotval Env -> Env [strat (3 2 1 0)] .
  op accessenv__ : Id Env -> Denotval [strat (2 1 0) memo] .
  op reserve-locn_ : Env -> Env .
  op get-locn_ : Env -> Locc .

  var L : Locc .
  var I : Id .
  var D : Denotval .
  var M : Idmap .
  eq : emptyenv = < ¦ , first-locn > .
  eq : updateenv I D < M , L > = < [ I D ] M , L > .
  eq : accessenv I < M , L > = M I .
  eq : reserve-locn < M , L > = < M , next-locn L > .
  eq : get-locn < M , L > = L .
endo

--- Syntactical domain of the abstract syntax
obj SYN-DOM is sorts Prog Block Decl Com Expr Bexp .
  pr QID .
  pr NAT .
  subsorts Nat < Expr .
  subsorts Id < Expr .

  op begin_end : Block -> Prog [prec 12] .
  op let_in_ : Decl Com -> Block [prec 11] .

  op _;_ : Decl Decl -> Decl [assoc prec 10] .
  op Const__ : Id Nat -> Decl [prec 8] .
  op Var_ : Id -> Decl [prec 8] .

  op _;_ : Com Com -> Com [assoc prec 10] .
  op _:=_ : Id Expr -> Com [prec 8] .
  op if_then_else_ : Bexp Com Com -> Com [prec 9] .
```

```
  op while_do_ : Bexp Com -> Com [prec 9] .
  op stop : -> Com .

  op _+_ : Expr Expr -> Expr [assoc prec 4] .

  op _eq_ : Expr Expr -> Bexp [prec 5] .
  op not_ : Bexp -> Bexp [prec 6] .
endo

--- Valuation function
obj VAL is
  pr SYN-DOM .
  pr STORE .
  pr ENV .
  ex CONT .

  op P[_] : Prog -> Answer .
  op K[_]__ : Block Env Ccont -> Ccont [strat (2 0) prec 7] .
  op D[_]_ : Decl Env -> Env [strat (2 0)] .
  op C[_]__ : Com Env Ccont -> Ccont [strat (2 0) prec 7] .
  op E[_]__ : Expr Env Econt -> Ccont [strat (2 0) prec 7] .
  op B[_]__ : Bexp Env Bcont -> Ccont [strat (2 0) prec 7] .
  op L[_]__ : Id Env Lcont -> Ccont [strat (2 0) prec 8] .

  var D D1 D2 : Decl .
  var K : Block .
  var C C1 C2 : Com .
  var E E1 E2 : Expr .
  var B : Bexp .
  var I : Id .
  var N : Nat .
  var En : Env .
  var Cc : Ccont .
  var Ec : Econt .
  var Bc : Bcont .
  var Lc : Lcont .
  var S : Store .

  eq : P[ begin K end ] = K[ K ] emptyenv (terminate normal) newstore .
  eq : K[ let D in C ] En Cc = C[ C ] (D[ D ] En) Cc .

  eq : D[ D1 ; D2 ] En = D[ D2 ] (D[ D1 ] En) .
  eq : D[ Var I ] En = updateenv I (get-locn En) (reserve-locn En) .
  eq : D[ Const I N ] En = updateenv I N En .

  eq : C[ C1 ; C2 ] En Cc = C[ C1 ] En (C[ C2 ] En Cc) .
  eq : C[ I := E ] En Cc S
       = L[ I ] En (& l . (E[ E ] En (& v . updatestore Cc l v))) S .
```

```
  eq : C[ if B then C1 else C2 ] En Cc
       = B[ B ] En (& t . (t => C[ C1 ] En Cc ; C[ C2 ] En Cc)) .
  eq : C[ while B do C ] En Cc
       = B[ B ] En
             (& t . (t => C[ C ] En (C[ while B do C ] En Cc) ; Cc)) .
  eq : C[ stop ] En Cc S = terminate stopped S .

  ceq : E[ I ] En Ec
       = Ec selecNat(accessenv I En) if isNat(accessenv I En) .
  ceq : E[ I ] En Ec S = Ec (access selecLoc(accessenv I En) S) S
                if isLoc(accessenv I En) .
  ceq : E[ I ] En Ec
       = terminate id-undefined if isUndefined(accessenv I En) .

  eq : E[ N ] En Ec = Ec N .
  eq : E[ E1 + E2 ] En Ec
       = E[ E1 ] En (& v1 . E[ E2 ] En (& v2 . Ec (v1 + v2))) .

  eq : B[ E1 eq E2 ] En Bc
       = E[ E1 ] En (& v1 . E[ E2 ] En (& v2 . (Bc equals(v1, v2)))) .
  eq : B[ not B ] En Bc = B[ B ] En (& t . Bc (not t)) .

  ceq : L[ I ] En Lc
       = Lc selecLoc(accessenv I En) if isLoc(accessenv I En) .
  ceq : L[ I ] En Lc = terminate id-use-err if isNat(accessenv I En) .
  ceq : L[ I ] En Lc
       = terminate id-undefined if isUndefined(accessenv I En) .

--- Auxiary equations for the application of continuations
  var V V' : SValue .
  var T : Tr .
  var L : Loc .
  var Cc1 Cc2 : Ccont .
  eq : (& l . (E[ E ] En (& v . updatestore Cc l v))) L
    = E[ E ] En (& v . updatestore Cc L v) .
  eq : (& v . updatestore Cc L v) V = updatestore Cc L V .
  eq : (& t . (t => Cc1 ; Cc2)) T = (T => Cc1 ; Cc2) .

  eq : (& v1 . E[ E ] En (& v2 . Ec (v1 + v2))) V
    = E[ E ] En (& v2 . Ec (V + v2)) .
  eq : (& v2 . Ec (V + v2)) V' = Ec (V + V') .

  eq : (& v1 . E[ E ] En (& v2 . Bc equals(v1, v2))) V
    = E[ E ] En (& v2 . Bc equals(V, v2)) .
  eq : (& v2 . Bc equals(V, v2)) V' = Bc equals(V, V') .
  eq : (& t . Bc (not t)) T = Bc (not T) .
endo
```

III. Example Runs

```
Welcome to OBJ3 Version .99
system built: (1988 3 15 15 5 55)
Copyright 1987 by the OBJ3 Group (KF, JAG, JPJ, JM, TW, CK, HK, AM)
OBJ3> in blok2
=========================================
in fun
Reading in file : "fun"

... ...

=========================================
obj ENV
=========================================
obj SYN-DOM
=========================================
obj VAL

OBJ3> reduce in VAL as :
  P[ begin
        let  Var 'x ; Const 'n 1 ; Var 'y
        in
            'x := 'n ;
            while 'x eq 10 do 'y := 'y + 1 ;
            'y := 'x + 1
     end ] .
reducing term: (P[ (begin (let ((Var 'x) ; ((Const 'n 1) ; (Var 'y))) in
 (('x := 'n) ; ((while ('x eq 10) do ('y := ('y + 1))) ; ('y := ('x +
 1))))) end) ])
reduction result Answer:
                 (< normal , ([ (L 1) 2 ] ([ (L 0) 1 ] newstore)) >)

OBJ3> reduce in VAL as :
  P[ begin
        let Var 'x ; Const 'n 10
        in
            'x := 'n ;
            if 'x eq 10 then stop
            else ('x := 'n + 1 ;
                   while not ('x eq 20) do 'x := 'x + 1) ;
            'x := 0
     end ] .
reducing term: (P[ (begin (let ((Var 'x) ; (Const 'n 10)) in (('x := 'n)
 ; ((if ('x eq 10) then stop else (('x := ('n + 1)) ; (while (not ('x eq
 20)) do ('x := ('x + 1))))) ; ('x := 0)))) end) ])
reduction result Answer: (< stopped , ([ (L 0) 10 ] newstore) >)
```

```
OBJ3> reduce in VAL as :
  P[ begin
        let Var 'x ; Const 'i 10
        in
            'x := 'i ;
            'i := 'x + 1
     end ] .
reducing term: (P[ (begin (let ((Var 'x) ; (Const 'i 10)) in (('x := 'i)
 ; ('i := ('x + 1)))) end) ])
reduction result Answer: (< id-use-err , ([ (L 0) 10 ] newstore) >)

OBJ3> reduce in VAL as :
  P[ begin
    let Var 'a
    in
        'a := 'n ;
        if 'a eq 10 then stop else 'a := 1
     end ] .
reducing term: (P[ (begin (let (Var 'a) in (('a := 'n) ; (if ('a eq 10)
 then stop else ('a := 1)))) end) ])
reduction result Answer: (< id-undefined , newstore >)
```

References

[BL 79] De Bakker, J. W.; Van Leeuwen, J. (eds): Foundations of Computer Science III: Part 2 Languages, Logic, Semantics, (draft), 1979.

[BW 87] Broy, M.; Wirsing, M.: On the Algebraic Definition of Programming Languages, *ACM Transactions on Programming Languages and Systems*, Vol.9, No.1, January 1987, pp 54-99.

[CH 87] Cohen, J.; Hickey, T.: Parsing and Compiling Using Prolog, *ACM Transactions on Programming Languages and Systems*, Vol.9, No.2, April 1987 pp 125-163.

[EM 85] Ehrig, H.; Mahr, B.: Fundamentals of Algebraic Specification 1 Equations and Initial Semantics, Springer-Verlag Berlin Heidelberg 1985.

[FGJM 87] Futatsugi, K.; Goguen, J.; Jouannaud, J.; Meseguer, J: Principles of OBJ2, *1987 ACM Principles of Programming Languages Symposium*, pp 52-66.

[G 87] Goguen, J. A.: Principles of Parameterized Programming, (draft) SRI International, Menlo Park CA, 1987.

[GM 86] Goguen, J. A.; Meseguer, J.: Semantics of Computation: Initial Algebra Semantics and Programming Language Paradigms, (draft), SRI International, Menlo Park CA, 1986.

[GM 88] Goguen, J. A.; Meseguer, J.: Order-Sorted Algebra I: Equational Deduction for Multiple Inheritance, Polymorphism, and Partial Operations, (draft), SRI International, Menlo Park CA, January 1988.

[GP 81] Goguen, J. A.; Parsaye-Ghomi, K.: Algebraic Denotational Semantics Using Parameterized Abstract Modules, *LNCS 107: Formalization of Programming Concepts* (eds: J. Diaz and I. Ramos), 1981, pp 292-309.

[GTW 78] Goguen, J. A.; Thatcher, J. W.; Wagner, E. G.; An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types, *Current Trends in Programming Methodology IV: Data Structuring* (R. Yeh, ed.), Prentice Hall (1978), pp 80-144.

[GTWW 77] Goguen, J. A.; Thatcher, J. W.; Wagner, E. G.; Wright, J. B.: Initial Algebra Semantics and Continuous Algebras, *Journal of ACM*, Volume 24 No.1, January 1977, pp 68-95.

[M 81] Mosses, P.: A Semantic Algebra for Binding Constructs, *LNCS 107: Formalization of Programming Concepts* (eds: J. Diaz and I. Ramos), 1981, pp 408-418.

[MW 88] Montenyohl, M.; Wand, M.: Correct Flow Analysis in Continuation Semantics, *The 15th ACM Symposium on Principles of Programming Languages*, January 1988, pp 204-218.

[NR 85] Nivat, M.; Reynolds, J. C.(eds): Algebraic Methods in Semantics, Cambridge University Press, 1985.

[PJs 97] Peyton Jones, S.: The implementation of Functional Programming Languages, Prentice Hall International, 1987.

[R 72] Reynolds, J.: Definitional Interpreters for Higher Order Programming, *Proc. ACM National Conference*, 1972, pp 717-739.

[Sc 85] Schmidt, D. A.: Detecting Global Variables in Denotational Specifications, *ACM Transactions on Programming Languages and Systems*, Vol. 7 No. 2, April 1985, pp 299-310.

[Sc 86] Schmidt, D. A.: Denotational Semantics A Methodology for Language Development, Allyn and Bacon, Inc. 1986.

[Sc 86a] Schmidt, D. A.: An Implementation from a Direct Semantics Definition, *LNCS 217: Programs as Data Objects* (eds: H. Ganzinger and N. D. Jones), 1986, pp 222-235.

[St 77] Stoy, J. E.: Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory, The MIT Press, 1977.

[W 82] Wand, M.: Deriving Target Code as a Representation of Continuation Semantics, *ACM Transactions on Programming Languages and Systems*, Vol.4, No.3, July 1982, pp 496-517.

ON THE ALGEBRAIC DENOTATIONAL SPECIFICATIONS OF PROGRAMMING LANGUAGE SEMANTICS

by

JUEMIN SUN

B.S., Fudan University (China), 1984

AN ABSTRACT OF A THESIS

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Dept. of Computing and Information Sciences

KANSAS STATE UNIVERSITY
Manhattan, Kansas

1988

ABSTRACT

This paper describes a method for giving algebraic denotational specifications of programming language semantics. The language used is OBJ3, a first-order parameterized algebraic specification language. The structure of specifications follows closely to the standard denotational semantics. Thus we are able to provide a tool to test the correctness of semantic definitions by executing OBJ3 specifications. Although our specifications are in denotational styles, we have the benefits of applying algebraic techniques: the definitions are highly structured with increased flexibility and easy verifiability. The major features of our semantic specifications are as follows: (1) We use the initial algebras of specifications as semantic domains. Domain constructions are regarded as set constructions. Methods of specifying compound domains are presented. (2) Curried operations in the original denotational semantics are specified in their decurried forms. (3) Since our specifications are first order, we have explored various ways of specifying higher order objects, including defunctionalizations and lambda conversions. The paper illustrates the methods with three complete semantic definitions: a direct semantics specification for a modest block-structured language, a continuation-based semantics specification for a similar block-structured language augmented with "stop", and a semantics specification for a LISP-like applicative language.